Harlen Costa Batagelo

Uma Arquitetura de Suporte a Interações 3D Integrada à GPU

Tese de Doutorado apresentada à Faculdade de Engenharia Elétrica e de Computação como parte dos requisitos para obtenção do título de Doutor em Engenharia Elétrica. Área de concentração: Engenharia de Computação.

Orientadora: Prof[a]. Dr.-Ing. Wu, Shin-Ting

Campinas, SP
2007

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

Batagelo, Harlen Costa
B31u Uma arquitetura de suporte a interações 3D integrada à GPU / Harlen Costa Batagelo. – Campinas, SP: [s.n.], 2007.

Orientador: Shin Ting Wu.
Tese (doutorado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Interfaces gráficas de usuário (Sistema de computador). 2. Computação gráfica. 3. Algoritmos de computador. I. Wu, Shin-Ting. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título

Título em Inglês:	A GPU-based architecture for supporting 3D interactions
Palavras-chave em Inglês:	Interaction, Direct manipulation, Programmable graphics hardware
Área de concentração:	Engenharia de Computação
Titulação:	Doutor em Engenharia Elétrica
Banca Examinadora:	João Luiz Dihl Comba, Jorge Stolfi, José Mario De Martino, José Raimundo de Oliveira e Léo Pini Magalhães
Data da defesa:	27/07/2007
Programa de Pós-Graduação:	Engenharia Elétrica

COMISSÃO JULGADORA - TESE DE DOUTORADO

Candidato: Harlen Costa Batagelo

Data da Defesa: 27 de julho de 2007

Título da Tese: "Uma Arquitetura de Suporte a Interações 3D Integrada à GPU"

Profa. Dra. Wu Shin-Ting (Presidente): _____

Prof. Dr. João Luiz Dihl Comba: _____

Prof. Dr. Jorge Stolfi: _____

Prof. Dr. José Raimundo de Oliveira: _____

Prof. Dr. José Mario De Martino: _____

Prof. Dr. Léo Pini Magalhães: _____

Resumo

Tendo como hipótese de que o controle preciso do movimento de um cursor constitui uma das técnicas elementares para as tarefas de manipulação direta 3D, esta tese propõe uma arquitetura de suporte a controles configuráveis dos movimentos de cursores em relação a modelos deformados em *hardware* gráfico. De forma integrada ao fluxo programável de visualização, a arquitetura calcula atributos de geometria diferencial discreta dos modelos processados, codificando tais atributos em *pixels* de *buffers* de renderização não visíveis. Mostramos, através de estudos de casos, que o uso desses atributos é suficiente para estabelecer uma correspondência entre o espaço discreto do modelo renderizado na tela e o espaço contínuo do modelo submetido ao fluxo de visualização. Isto permite que os cursores sejam posicionados de forma consistente com aquilo que o usuário está visualizando, proporcionando uma interação mais acurada. Testes de desempenho e robustez são conduzidos para validar a arquitetura. Uma biblioteca de funções que encapsula a arquitetura é apresentada, juntamente com exemplos de tarefas de manipulação direta 3D implementadas através dela.

Palavras-chave: Interação, manipulação direta, GPU.

Abstract

Based on the hypothesis that the precise control of the motion of a cursor constitutes one of the elementary techniques for 3D direct manipulation tools, this thesis proposes an architecture for supporting a configurable control of the motion of cursors with respect to models deformed on graphics hardware. Integrated with the actual programmable rendering pipeline, the architecture computes discrete differential geometric attributes of the processed models and encodes such attributes in pixels of off-screen render buffers. We show, through case studies, that these attributes are sufficient to establish a correspondence between the discrete space of the model rendered on the screen and the continuous space of the model submitted to the rendering pipeline. As a result, the cursors can be positioned consistently with what the user is actually viewing, thus providing a more accurate interaction. Efficiency and reliability tests are conducted to validate the architecture. A library of functions that encapsulates the architecture and examples of 3D direct manipulation tasks implemented with it are also presented.

Keywords: Interaction, direct manipulation, GPU.

Agradecimentos

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), pelo auxílio financeiro fornecido através do processo 141685/2002-6.

À Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), cujo auxílio financeiro através do processo 03/13090-6 foi fundamental para a aquisição de infra-estrutura utilizada no desenvolvimento da tese.

Aos meu pais.

Sumário

Lista de Figuras

Lista de Tabelas

Trabalhos publicados pelo autor

1. H.C. Batagelo, S.T. Wu, "Application-independent 3D interaction using geometry attributes computed on the GPU." In: *Proceedings of the 20th Brazilian Symposium on Computer Graphics and Image Processing* (Belo Horizonte, MG, Brasil, Outubro de 2007), IEEE CS Press. pg. 19-26.

   Este artigo resume o conteúdo do capítulo 5 desta tese. Propomos uma arquitetura de suporte à implementação de tarefas de manipulação direta 3D com base no pressuposto de que atributos de geometria diferencial e atributos definidos pelo usuário disponibilizados para cada *pixel* do modelo renderizado são suficientes para a realização de diversas tarefas de interação 3D. Também mostramos que, através da integração da arquitetura proposta com a arquitetura das atuais GPUs, deformações de geometria no fluxo de visualização podem ser levadas em consideração de forma eficiente.

2. H.C. Batagelo, S.T. Wu. "Estimating curvatures and their derivatives on meshes of arbitrary topology from sampling directions." *The Visual Computer*, Setembro de 2007, Springer Berlin / Heidelberg, 23:9-11, pg. 803-812.

   Este trabalho apresenta uma técnica de estimativa de propriedades de geometria diferencial de segunda e terceira ordem sobre malhas geométricas de topologia arbitrária. A técnica é adequada à implementação eficiente nas atuais GPUs e utiliza as informações de vetores normais já existentes de modo a produzir resultados acurados em malhas com ruído ou amostradas de forma irregular. Na arquitetura proposta neste trabalho de tese, esta técnica é utilizada para estimar propriedades geométricas de forma independente da aplicação, após as modificações de atributos dos vértices na GPU. Uma descrição detalhada é mostrada no capítulo 4.

3. H.C. Batagelo, S.T. Wu. "What You See Is What You Snap: Snapping to Geometry Deformed on the GPU." In: *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games* (Washington, DC, EUA, Abril de 2005), ACM Press, pg. 81-86.

   Este trabalho apresenta a proposta, detalhada no capítulo 3, de utilizar dados geométricos armazenados para cada *pixel* da imagem de modo a realizar tarefas de manipulação direta 3D sobre modelos deformados arbitrariamente em *hardware* gráfico programável. As tarefas suportadas incluem seleção, posicionamento restrito de um cursor 3D a uma superfície e pintura 3D sem restrição de posicionamento. A arquitetura implementada neste trabalho de tese amplia o número de tarefas de interação suportadas, estimando na GPU propriedades de geometria diferencial tais como vetor normal, vetores tangentes, curvaturas e suas derivadas.

4. S.T. Wu, M. Abrantes, D. Tost, H.C. Batagelo. "Picking and Snapping for 3D Input Devices." In: *Proceedings of the 16th Brazilian Symposium on Computer Graphics and Image Processing* (São Carlos, SP, Brasil, Outubro de 2003), IEEE CS Press, pg. 140-147.

   Este trabalho introduz a idéia de realizar tarefas de seleção e posicionamento restrito de um cursor 3D com base em um conjunto mínimo e suficiente de propriedades geométricas locais do modelo: posição 3D e vetor normal. O trabalho explora a possibilidade de obter tais propriedades de modo independente da aplicação, *i.e.*, sem requerer algoritmos específicos de seleção baseados em *ray picking*. Nesta tese, esta idéia serviu de inspiração para propor as hipóteses mostradas no capítulo 1, sobre as quais a arquitetura de suporte a interações 3D é desenvolvida.

Glossário

GPU	-	Unidade de processamento gráfico (*Graphics Processing Unit*).
WYSIWYG	-	Acrônimo, em inglês, da expressão "O que você vê é o que você tem" (*What You See Is What You Get*).
CAD	-	Projeto auxiliado por computador (*Computer Aided Design*).
VRML	-	Linguagem para modelagem de realidade virtual (*Virtual Reality Modeling Language*).
SGI	-	Silicon Graphics, Inc.
IRIS	-	Sistema de Imageamento *Raster* Integrado (*Integrated Raster Imaging System*).
ARB	-	Comitê de revisão de arquitetura da biblioteca OpenGL (*Architecture Review Board*).
RHW	-	Valor recíproco da coordenada homogênea W (*Reciprocal Homogeneous W*).
BRDF	-	Função de distribuição de reflectância bidirecional (*Bidirectional Reflectance Distribution Function*).
MIMD	-	Arquitetura de *hardware* baseada no uso de múltiplas instruções para o processamento de múltiplos dados (*Multiple Instruction Multiple Data*).
SIMD	-	Arquitetura de *hardware* baseada no uso de uma única instrução para o processamento de múltiplos dados (*Single Instruction Multiple Data*).
AGP	-	Padrão de barramento de comunicação entre a CPU e o *hardware* gráfico (*Advanced Graphics Port*).
PCI	-	Padrão de barramento de comunicação entre a CPU e periféricos (*Peripheral Component Interconnect*).
FPS	-	Quadros por segundo (*Frames Per Second*).

Capítulo 1

Introdução

Em pouco mais de 20 anos, e especialmente na última década, o *hardware* gráfico compatível com as chamadas *APIs GL*, tais como IrisGL [McLendon, 1992], Glide [3Dfx, 1997], OpenGL [Shreiner et al., 2005] e Direct3D [Microsoft, 2006], evoluiu de forma muito significativa, a ponto de redefinir o modo como esse tipo de equipamento é utilizado em aplicações de Computação Gráfica. Amplamente difundidas em computadores pessoais, as atuais unidades de processamento gráfico, também chamadas de GPUs (*Graphics Processing Units*) são dotadas de poder computacional e flexibilidade tais que extrapolam seu emprego original de servir como processadores dedicados ao processamento para síntese de imagens. Apesar disso, até então poucos esforços foram empregados na análise da viabilidade do uso de *hardware* gráfico em tarefas de interação 3D, seja para acelerar o processamento de interação, seja para obter resultados mais consistentes com relação às modificações de geometria que as atuais GPUs podem produzir durante o processamento no fluxo de visualização.

As atuais técnicas de interação 3D, segundo o estilo de interação de *manipulação direta* [Shneiderman, 1983], são geralmente baseadas em algoritmos geométricos propostos entre as décadas de 1960 a 1990 e não consideram a possibilidade da existência de processamento de geometria em *hardware gráfico* introduzido pelas GPUs a partir de 2000. O impacto que as GPUs atuais provoca na realização de interações de manipulação direta tem sido subestimado, não obstante a existência de dificuldades práticas que surgiram nesses últimos anos e que tendem a se tornar cada vez mais significativas. Em particular, atualmente o *hardware* gráfico é capaz de modificar arbitrariamente as primitivas geométricas submetidas pela CPU, tanto com relação aos vértices como fragmentos de *pixels* produzidos pelo rasterizador, de sorte que os modelos exibidos ao usuário podem ser muito diversos daqueles sobre os quais os algoritmos de interação utilizam na CPU. Essa discrepância implica a produção de interações inconsistentes com aquilo que está sendo visualizado, podendo tornar inviável o uso dos recursos de aceleração de *hardware* em aplicações que dependem de manipulação direta 3D. Atualmente não há solução satisfatória para esse problema e as alternativas práticas mais

utilizadas consistem em apenas reproduzir todo o processamento da geometria na CPU.

Esta tese procura reduzir essa lacuna existente entre o uso de técnicas de interação e o uso de técnicas de animação, modelagem e visualização aceleradas por GPUs. Em especial, propomos uma solução eficiente e geral para o problema de manipulação direta 3D em geometria modificada em processadores de vértices do *hardware* gráfico, mas que também fornece suporte básico a geometria modificada em processadores de fragmentos. Nossa proposta está baseada na hipótese de que é possível calcular na própria GPU um conjunto mínimo de atributos geométricos necessários e suficientes para tarefas de manipulação direta baseadas no uso de seleção e posicionamento através de dispositivos apontadores 2D, e tornar esses atributos disponíveis à aplicação sem requerer que a geometria deformada esteja necessariamente armazenada em um banco de dados da cena na memória do sistema.

O restante deste capítulo está organizado da seguinte forma: A seção 1.1 apresenta as principais motivações para a realização deste trabalho. A seção 1.2 apresenta as hipóteses sobre as quais construímos uma proposta de resolução dos problemas que nos motivaram, e um resumo das principais contribuições derivadas dessa proposta. O capítulo é concluído na seção 1.3 com a apresentação da organização da tese.

1.1 Motivação

O mais recente impacto gerado pelo uso de *hardware* gráfico nas diversas áreas da Computação Gráfica se deve fundamentalmente ao advento de GPUs programáveis. Em geral, este tipo de *hardware* é composto de dois processadores programáveis: um processador de vértices, para o processamento de dados vetoriais, e um processador de fragmentos, para o processamento de dados *raster*. Para a mais recente geração de placas gráficas compatíveis com Direct3D 10 [Blythe, 2006], um "processador de geometria" (*geometry processor*) é acrescentado, e possibilita o acesso a informações sobre uma primitiva completa, tais como topologia e atributos de todos os seus vértices. Em comum, os processadores de vértices, de geometria e de fragmentos trabalham em fluxos de processamento paralelo interno e de forma assíncrona em relação à CPU.

Em *hardware* gráfico atual, é comum a existência de arquiteturas capazes de processar centenas de *pixels* de forma simultânea no processador de fragmentos [Kilgariff and Fernando, 2005]. Em razão desse paralelismo, e aliado ao poder computacional e flexibilidade do conjunto de instruções para programação dos processadores, o *hardware* gráfico atual tem sido utilizado como um processador de fluxo de propósito geral, *i.e.*, não necessariamente relacionado à síntese de imagens ou até mesmo aos problemas tratados em Computação Gráfica. Por exemplo, em um projeto sobre a implementação de um algoritmo de ordenação bitônica na GPU, Govindaraju *et al.* [Govindaraju et al., 2003] cons-

tataram que a ordenação de 16 milhões de registros de um banco de dados poderia ser realizada em apenas 2 segundos em uma placa gráfica NVIDIA GeForce 7800 GTX. O mesmo processamento era realizado em no mínimo 15 segundos usando a função `qsort` (ANSI C) otimizada no compilador Intel C++ com instruções SSE (*Streaming SIMD Extensions*), e sendo executado numa CPU Intel Pentium 4 de 3.2 GHz utilizando a tecnologia *Hyper-Threading*.

Hoje, o maior mercado consumidor de *hardware* gráfico programável e principal responsável pelo impulso do desenvolvimento de novas tecnologias é o mercado de entretenimento digital (*e.g.*, jogos para computadores e *video games*). Para essas aplicações, o uso não convencional das GPUs tem sido comumente concentrado na realização de tarefas de animação e modelagem geométrica em malhas triangulares. Embora anteriormente a GPU fosse utilizada apenas para a aceleração das etapas de transformação geométrica, cálculo de iluminação e rasterização, verificou-se que tarefas mais complexas de processamento de geometria tais como *mesh skinning* [Dempski, 2002], simulação de ondas no oceano [Isidoro et al., 2002], modelagem procedural de terrenos [Green, 2002], simulação de tecidos [Zeller, 2005] e mapeamento de deslocamento [Kryachko, 2005] poderiam ser realizadas totalmente na GPU. Essa estratégia, além de aproveitar melhor o paralelismo entre a CPU e a GPU, também se beneficia da ampla largura de banda existente no acesso a recursos armazenados em memória de vídeo local: de 8 GB/s do barramento PCI Express 16X para mais de 50 GB/s da memória de vídeo local em uma placa NVIDIA GeForce 7900 GTX [Kilgariff and Fernando, 2005]. Atualmente, o barramento de uma placa NVIDIA GeForce 8800 Ultra tem uma largura de banda de 103.7 GB/s.

Uma ilustração do acesso a recursos da memória de vídeo e memória do sistema é mostrada na figura 1.1. Nas técnicas citadas, a CPU envia apenas uma vez um modelo geométrico estático para a GPU. Este modelo é armazenado em memória de vídeo local e, para cada quadro de exibição, ele é lido, processado e renderizado pelo *hardware* gráfico sem intervenção da CPU.

A capacidade de deformar geometria na GPU sem intervenção da CPU introduz sérias complicações para a implementação de algoritmos de interação 3D que trabalham com tais modelos. Tradicionalmente, o processamento necessário para executar tarefas de manipulação direta tais como seleção (*picking*) e posicionamento com restrição (*snapping*) usando dispositivos apontadores, é realizado inteiramente na CPU. Esse processamento é feito com base em informações fornecidas pelo sistema de janelas (*e.g.*, eventos dos dispositivos de entrada sobre os controles de interface 2D) e com base no modelo geométrico armazenado em um banco de dados sobre a cena 3D na memória do sistema. Um diagrama conceitual ilustrando essas relações é mostrado na figura 1.2.

Se a GPU é utilizada para executar técnicas de animação e modelagem geométrica, instâncias dos modelos geométricos devem ser armazenadas em memória de vídeo local. Porém, se as instâncias deformadas na GPU não são propagadas para o modelo original armazenado na memória do sistema,

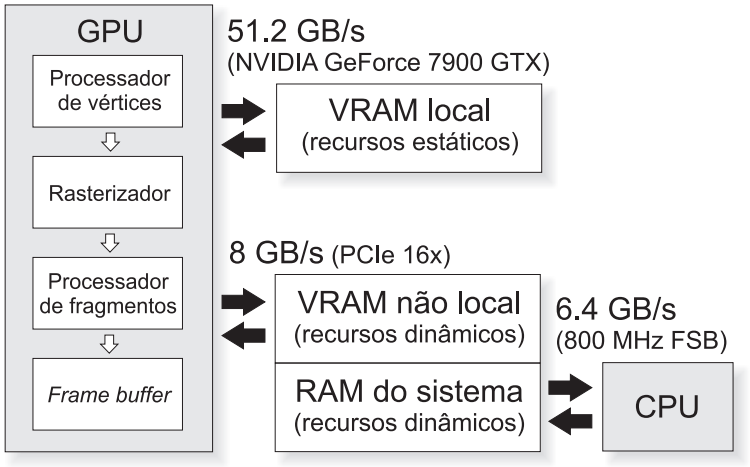


Fig. 1.1: Largura de banda da transferência de recursos armazenados na memória de vídeo e na memória do sistema.

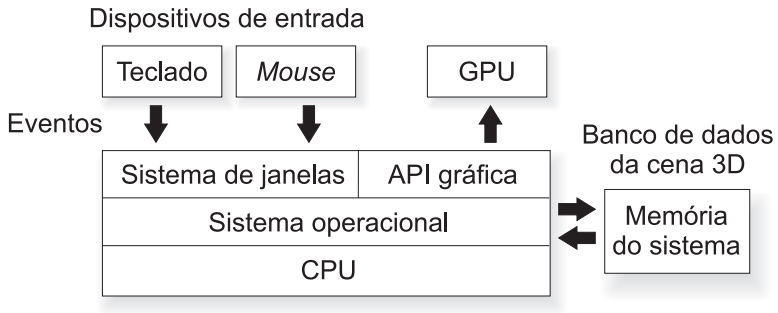


Fig. 1.2: Diagrama conceitual do fluxo de eventos de entrada e saída em um paradigma tradicional de interação.

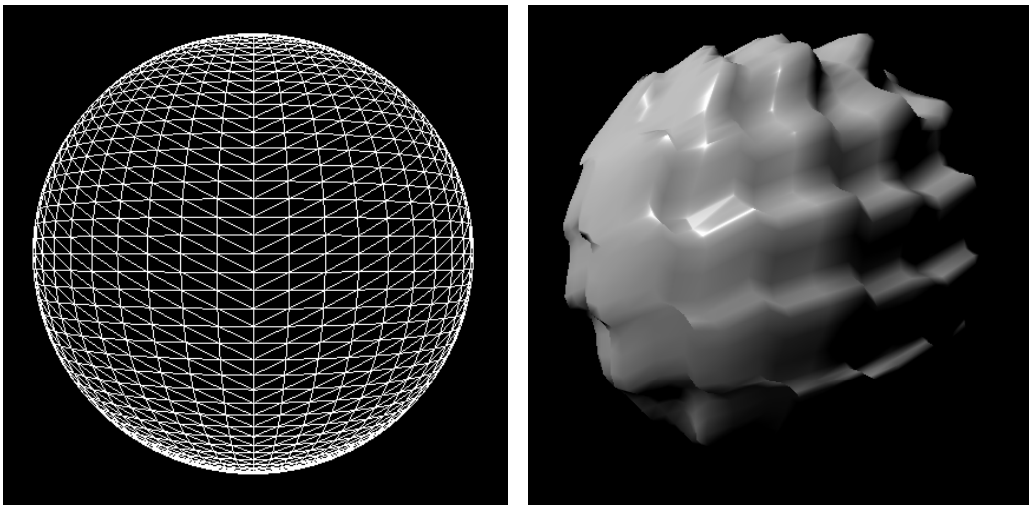


Fig. 1.3: Deformação dos vértices de uma esfera na GPU usando uma função senoidal de deslocamento. Esquerda: geometria original submetida pela CPU, em *wireframe*. Direita: imagem obtida após processamento da geometria na GPU.

podem ocorrer inconsistências entre as ações de interação realizadas pelo usuário e o resultado visual obtido. Por exemplo, suponhamos que o modelo geométrico original seja uma esfera e que, na instância armazenada em memória de vídeo, seus vértices sejam deslocados ao longo dos vetores normais de acordo com uma função de deslocamento na GPU. O resultado visual não terá mais a aparência de uma esfera (Figura 1.3). Entretanto, quando o usuário tentar interagir com este novo modelo observado, todo o processamento de interação ainda estará sendo realizado na CPU, com base no modelo não deformado, *i.e.*, uma esfera. Inconsistências deste tipo poderão ser evidentes mesmo em tarefas simples como apontar e selecionar um ponto específico do modelo usando um dispositivo apontador que controla um cursor 2D.

As técnicas de animação e modelagem geométrica citadas anteriormente (*e.g.*, *mesh skinning*, modelagem procedural de terrenos e mapeamento de deslocamento) podem modificar os atributos de cada vértice por meio de transformações que não preservam as propriedades locais de geometria diferencial do modelo representado, tais como a área e a curvatura. Assim, a propagação dessas mudanças para o modelo na CPU geralmente não pode ser feita pelo uso de uma única transformação afim sobre toda a geometria, o que seria realizado através da multiplicação do vetor de cada ponto 3D em coordenadas homogêneas por uma matriz de transformação afim. Por este motivo, a solução mais comum para realizar interações consistentes de geometria deformada na GPU consiste em executar todo o código de deformação também na CPU, de forma paralela ao processamento para renderização. Infelizmente, essa estratégia não é satisfatória pois anula todos os benefícios do uso da GPU para liberar a CPU do processamento de deformação de geometria. Ao contrário, o mesmo proces-

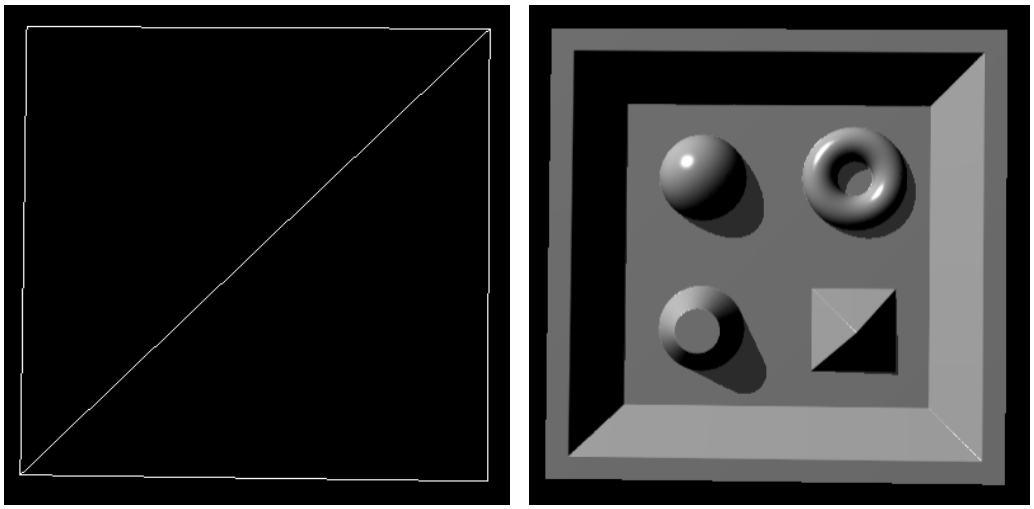


Fig. 1.4: Deformação de fragmentos na GPU usando a técnica de *relief mapping*. Esquerda: geometria original submetida pela CPU, em *wireframe*. Direita: imagem obtida após processamento na GPU.

samento de deformação é realizado duas vezes, exigindo duplicação de código. Além disso, requer a manutenção dos dados da geometria original na memória do sistema, gerando duplicação de dados.

É cada vez mais comum o uso de técnicas de mapeamento de detalhes 3D para aumentar de forma eficiente o aparente nível de detalhes dos modelos geométricos. Nessas técnicas, modelos simples compostos a partir de apenas um triângulo podem simular a aparência de geometrias complexas compostas de vários milhões de triângulos. Isso é obtido através da modificação dos fragmentos de *pixels* gerados pelo rasterizador, em oposição à modificação ou adição de vértices. Técnicas como *normal mapping* [Krishnamurthy and Levoy, 1996], *parallax occlusion mapping* [Tatarchuk, 2006] e *relief mapping* [Policarpo et al., 2005] modificam os atributos de cada fragmento no processador de fragmentos e, dependendo da sofisticação de cada técnica, podem gerar detalhes 3D com paralaxe, oclusão e silhuetas corretas. O resultado visual geralmente aparenta ser muito mais complexo e diverso da geometria original submetida à GPU pela CPU (Figura 1.4).

Da mesma forma que na deformação com base nos vértices citada anteriormente, se o processamento de interação é realizado com o modelo original na CPU, as modificações dos fragmentos não serão levadas em conta, novamente gerando inconsistências. Transformações decorrentes do mapeamento de detalhes também poderiam ser simuladas no modelo armazenado na memória do sistema. Porém, nesse caso seria necessário simular na CPU todo o fluxo de processamento de fragmentos da GPU, o que é impraticável como solução eficiente para o problema exposto, além de anular os benefícios do uso da GPU para processamento paralelo, como já citado para o caso da modificação de atributos dos vértices.

Como então interagir de forma consistente e ao mesmo tempo eficiente com um modelo geométrico modificado de forma arbitrária na GPU, considerando que essas modificações podem ocorrer tanto com relação aos vértices como fragmentos produzidos pelo rasterizador?

A indústria de *hardware* gráfico, em um trabalho conjunto com os desenvolvedores das APIs, têm sinalizado como tendência para as próximas gerações de *hardware* gráfico uma flexibilidade cada vez maior para criar ou destruir primitivas geométricas dentro da própria GPU, sem que estas precisem necessariamente existir na memória do sistema. Indicativos desta característica já são vistos nas atuais GPUs compatíveis com o chamado *Shader Model 4.0*[1] [Blythe, 2006], e implica a não obrigatoriedade da existência de um modelo na CPU com a mesma topologia do modelo existente na memória de vídeo local. Sabendo disso, é desejável que uma solução para o problema da realização de interações consistentes leve em consideração a possível inexistência da geometria na memória do sistema. Tal solução deve ainda minimizar a duplicação de código e dados na CPU, de modo a manter os benefícios obtidos pelo uso da GPU nos casos citados.

Idealmente, o tratamento de eventos de entrada e saída para interação com modelos geométricos processados na GPU não deveria demandar qualquer tipo de processamento na CPU. Por outro lado, essa possibilidade de completo desacoplamento do processamento entre a CPU e a GPU não pode ser concretizada senão através da execução das tarefas de um sistema de janelas na placa gráfica. Como não existe no momento uma arquitetura de *hardware* que suporte esta possibilidade, devemos considerar que a CPU ainda desempenha um papel fundamental no tratamento dos eventos de entrada e saída através do sistema de janelas. Desse modo, o processamento na CPU continua existindo, seja para disparar o processamento de fluxo da GPU através de chamadas de funções de renderização, seja para utilizar os resultados produzidos na GPU para realizar a realimentação da interação.

Os itens abaixo sintetizam as principais motivações deste trabalho:

- Resolução de um problema prático para o qual não há solução satisfatória atualmente, e que tende a se tornar cada vez mais evidente: manipulação direta 3D consistente com primitivas geométricas processadas de forma arbitrária na GPU, tanto com relação aos vértices como com relação aos fragmentos produzidos pelo rasterizador. Deve ser ainda considerada a possibilidade de inexistência de dados dessas primitivas na memória do sistema, embora o processamento de eventos provenientes dos periféricos seja realizado na CPU pelo sistema de janelas.

- Necessidade de disponibilizar ferramentas de edição WYSIWYG (*What You See Is What You Get*) em aplicações de autoria de recursos gráficos, tais como aplicativos de modelagem de cenas 3D com aplicação de *shaders* de deformação de geometria em tempo real. No ambiente

[1] O termo *Shader Model* (modelo de *shader*) é utilizado para designar a versão do ambiente de execução do processador de vértices e fragmentos das GPUs programáveis. Veja a seção 2.3.4.

de produção, o uso de tais ferramentas pode ajudar a reduzir a lacuna existente entre o sistema no qual o artista gráfico trabalha, e o sistema utilizado pelos programadores para produzir o produto final que utiliza os recursos gráficos. Aplicativos deste tipo, como o Right Hemisphere Deep Creator/Deep Paint 3D [Hemisphere, 2007], Pixologic ZBrush [Pixologic, 2007] e Interactive Effects Amazon Paint [Effects, 2007], podem se beneficiar de ferramentas de edição capazes de atuar diretamente sobre modelos modificados na GPU ao fornecer ao usuário uma interface mais amigável e que atende o paradigma WYSIWYG. Neste caso, o usuário pode trabalhar com o modelo na forma como ele será apresentado na aplicação final.

- Capacidade de utilizar o poder computacional da GPU para realizar processamento de interação de forma paralela à CPU. Tal característica é especialmente importante em jogos e outras aplicações de processamento gráfico intensivo que requerem o uso de manipulação direta 3D como forma de interação com os objetos da cena. Ao realizar o processamento de interação em *hardware* gráfico, a CPU fica livre para executar outras tarefas de forma paralela (*e.g.*, tarefas da lógica do jogo). Além disso, o alto poder computacional das GPUs pode reduzir o próprio processamento da interação, mesmo no caso em que as primitivas geométricas não sofrem deformações de vértices ou fragmentos.

- Em aplicações de CAD (*Computer Aided Design*), diferentes representações de modelos geométricos (*e.g.*, representação explícita, implícita ou paramétrica) requerem algoritmos distintos de interseção entre raios e superfícies (*ray picking*) para determinar os pontos de seleção. Por outro lado, a visualização desses modelos com aceleração em *hardware* gráfico é comumente feita através de representações formadas de primitivas suportadas pelas GPUs atuais, a saber, triângulos, pontos e linhas.[2] Para aumentar a flexibilidade do algoritmo de interação com relação a diferentes tipos de geometria, a GPU pode ser utilizada para realizar o cálculo de interação sobre a representação submetida pela CPU. O resultado pode ser utilizado diretamente para realimentação visual, ou como aproximação para a solução exata, ou ainda como passo inicial de um processo iterativo de refinamento da interseção com a geometria exata.

1.2 Hipótese e contribuição

Esta tese defende a hipótese de que a atual arquitetura de *hardware* gráfico programável é flexível o suficiente para processar atributos geométricos necessários para a realização de tarefas de manipula-

[2]Já foram propostas técnicas de traçado de raios em tempo real através do uso da GPU como um processador de propósito geral [Purcell et al., 2002, Purcell, 2004]. Embora o paradigma de varredura ainda seja mais eficiente em *hardware* atual, propostas de processadores dedicados para traçado de raios também já foram apresentadas [Woop et al., 2005].

ção direta com dispositivos apontadores 2D. Em particular, essa hipótese é aplicada a todas as tarefas de manipulação direta constituídas com base nas tarefas básicas de posicionamento, seleção e restrição do posicionamento do cursor. Assim, tarefas como apontar e selecionar modelos, ou restringir a movimentação de um cursor a uma superfície, podem ser realizadas com atributos da geometria calculados inteiramente pela GPU, e interpretados pela CPU a partir da leitura dos *pixels* que contém a superfície renderizada. Conjeturamos ainda que é possível definir um conjunto mínimo de atributos geométricos e não geométricos capazes de suportar tais tarefas de manipulação. Esse conjunto é composto pelas propriedades de geometria diferencial discreta e atributos definidos pela aplicação para cada vértice ou fragmento do modelo.

Com base nessas hipóteses, apresentamos a proposta e implementação de uma arquitetura de *software* para o suporte à interação 3D explorando a atual arquitetura de *hardware* programável das GPUs. Nessa arquitetura de interação, atributos geométricos das primitivas renderizadas e, opcionalmente, atributos não geométricos definidos pela aplicação, são processados pela GPU e codificados na própria imagem gerada, para cada *pixel*. A CPU pode então decodificar os atributos do *pixel* coincidente com o ponto corrente associado ao cursor do dispositivo apontador 2D e assim restaurar as propriedades da geometria 3D nesse ponto. Como resultado, não há duplicação de processamento de deformação, e resultados visuais consistentes podem ser obtidos para geometrias modificadas tanto em seus atributos de vértices como atributos de fragmentos. Isso inclui as deformações de vértices que ocorrem nas técnicas de animação e modelagem geométrica citadas na seção 1.1, e pode incluir as modificações realizadas por técnicas de mapeamento de detalhes 3D ou resultantes da execução de filtros de pós-processamento de imagem. Tal arquitetura também é compatível com a possibilidade de inexistência do modelo geométrico deformado na CPU, pois os atributos geométricos podem ser obtidos apenas dos dados submetidos à GPU após o processamento realizado no fluxo de visualização.

Apresentamos também a implementação de uma série de tarefas de manipulação direta em geometria deformada na GPU usando a arquitetura proposta: restrição (de um cursor 3D controlado por um cursor 2D) a superfícies, restrição a vértices e arestas com base em funções de gravidade, restrição a pontos de superfície coincidentes com características da imagem, pintura 3D e escultura de modelos processados com mapeamento de detalhes 3D. A implementação dessas técnicas procura validar a proposta da arquitetura de interação, mostrar sua eficiência, robustez e generalidade.

Em resumo, a principal contribuição desta tese é a proposta de um conjunto de propriedades geométricas diferenciais e implementação de uma arquitetura de interação 3D que fornece suporte a tarefas de manipulação direta usando dispositivos apontadores 2D em modelos geométricos com atributos modificados na GPU. Entre as contribuições secundárias obtidas ao longo da implementação da arquitetura de interação, destacamos a implementação de um algoritmo na GPU para a estimativa de propriedades de geometria diferencial de malhas triangulares arbitrárias. Em especial, para cada

vértice estimamos vetores normais, vetores tangentes alinhados de acordo com a parametrização das coordenadas de textura, tensor de curvatura, curvaturas principais, direções principais e tensor de derivada de curvatura. Outras quantidades, como a curvatura média e Gaussiana, são extraídas dessas propriedades calculadas. Isso assegura, ao desenvolvedor, a transparência do tratamento das deformações sofridas pela geometria submetida à GPU no processador de vértices. Uma outra contribuição é a implementação da arquitetura como uma biblioteca de funções C++ com OpenGL e Direct3D. Tal biblioteca encontra-se disponível em Batagelo and Wu [2007a].

1.3 Visão geral

Os capítulos seguintes estão organizados da seguinte forma. No capítulo 2 apresentamos uma revisão bibliográfica sobre manipulação direta, um histórico sobre a evolução das arquiteturas de interface gráfica 3D com suporte a manipulações diretas, e um histórico da evolução do *hardware* gráfico compatível com APIs gráficas GL. Também apresentamos detalhes da atual arquitetura programável das GPUs, e mostramos como as GPUs podem ser utilizadas para realizar processamento de propósito geral, uma vez que essa possibilidade é explorada pela arquitetura proposta.

No capítulo 3, mostramos como diferentes técnicas de manipulação direta 3D utilizando dispositivos apontadores 2D podem ser realizadas a partir da leitura de um conjunto suficiente de atributos geométricos (propriedades de geometria diferencial) e não geométricos (valores definidos pela aplicação) armazenados em cada *pixel* do modelo renderizado. Esta possibilidade, aliada à flexibilidade das atuais GPUs, serve de motivação para apresentarmos a proposta de arquitetura de suporte à tarefas de interação 3D capaz de explorar a própria GPU para computar de forma eficiente esses conjuntos de atributos. O capítulo inclui ainda a apresentação de conceitos de geometria diferencial utilizados ao longo deste trabalho de tese.

No capítulo 4 apresentamos métodos de estimativa de propriedades de geometria diferencial discreta em *hardware* gráfico, com base na capacidade de utilizar as atuais GPUs como processadores de propósito geral. Esses algoritmos são utilizados na arquitetura para calcular em tempo real as propriedades geométricas dos modelos tratados. Resultados de testes de eficiência e robustez desses algoritmos também são apresentados neste capítulo. Os códigos dos *shaders* utilizados para realizar as estimativas são apresentados nos apêndices A e B.

A proposta da arquitetura de suporte à tarefas de interação 3D é apresentada no capítulo 5. Inicialmente mostramos os requisitos sobre os quais nos baseamos para propor tal arquitetura e então delimitamos seu escopo de atuação com relação às tarefas de manipulação direta. Neste capítulo também detalhamos a idéia básica de armazenar atributos geométricos ou não geométricos para cada *pixel* da imagem renderizada, e mostramos a especificação da interface de entrada e saída entre a

arquitetura e a aplicação, bem como um procedimento geral de uso. O detalhamento da interface de programação da arquitetura é apresentado no apêndice C.

No capítulo 6 apresentamos os resultados dos testes de desempenho da implementação da arquitetura proposta. De modo a demonstrar as características de generalidade e reusabilidade da arquitetura, também apresentamos a implementação de diversas tarefas de manipulação direta em geometria deformada na GPU com relação tanto aos vértices como aos fragmentos.

As considerações finais da tese, incluindo uma discussão sobre as limitações da arquitetura e os trabalhos futuros, são dadas no capítulo 7.

Capítulo 2

Revisão bibliográfica

Neste capítulo apresentamos uma revisão dos conceitos de interação utilizados nos próximos capítulos, relacionamos trabalhos anteriores sobre implementação de tarefas básicas de manipulação direta 3D e apresentamos um histórico da evolução das arquiteturas de interface gráfica e *hardware* gráfico. Na seção 2.1, apresentamos os conceitos de técnicas de interação, tarefas de interação básicas e compostas, e restrições de posicionamento. Em seguida, na seção 2.2, definimos o estilo de interação por manipulação direta. De modo a situar nossa proposta ante a evolução das tecnologias de interação e visualização, também apresentamos nesta seção um histórico das arquiteturas de interface gráfica com suporte a manipulações diretas 3D e, na seção 2.3, um histórico da evolução do *hardware* gráfico. A seção 2.4 descreve a arquitetura das atuais GPUs programáveis e como elas podem ser empregadas para realizar processamento de propósito geral, uma vez que este tipo de processamento é utilizado para estimar propriedades de geometria diferencial dos modelos tratados. Finalizamos com as considerações finais na seção 2.5.

2.1 Técnicas e tarefas de interação

O desenvolvimento de interfaces gráficas está associado ao tratamento de dois problemas fundamentais: interação e visualização. A interação é determinada pela reciprocidade de ações do usuário e do sistema gráfico. Eventos são disparados a partir de dispositivos de entrada acionados pelo usuário e são interpretados pelo sistema de modo a produzir ações correspondentes de modificação dos objetos de interesse. Por sua vez, a visualização é a forma de saída produzida pelo sistema através de dispositivos que tornam possível a percepção visual desses objetos e/ou ações produzidas.

Segundo Foley et al. [1990], a forma como os dispositivos de entrada são utilizados pelo usuário para entrar informações semanticamente válidas no contexto de uma aplicação de computador é definida pelas *técnicas de interação*. As técnicas de interação formam os blocos básicos de cons-

trução de uma interface e definem como uma tarefa é realizada pelo usuário. Menus de opções, consoles de linha de comando, botões e reconhecimento de voz são exemplos dessas técnicas.

Em interfaces gráficas, as representações gráficas e o comportamento das componentes envolvidas nas técnicas de interação são, em conjunto, chamadas de *widgets*. Por exemplo, em uma técnica de interação através de botões, os *widgets* são representações gráficas desses botões e comportamento de cada botão em relação às ações dos usuários. A aparência dos *widgets* é alterada pela aplicação de acordo com o tratamento dos eventos dos dispositivos de entrada, de modo que o usuário tenha um retorno visual adequado sobre a ação que está sendo realizada.

Os tipos fundamentais de informações fornecidas à aplicação por meio de técnicas de interação são classificados segundo as *tarefas de interação*. As tarefas de interação definem *quais* informações serão fornecidas à aplicação, enquanto que as *técnicas de interação* definem *como* estas informações serão fornecidas. Uma tarefa de interação é *básica* se o seu resultado corresponde a uma unidade indivisível de informação. Cinco tarefas básicas mais usuais são:

- **Posicionamento.** Utilizada para fornecer à aplicação a posição de um ponto, tal como o ponto de avaliação de um cursor 2D ou 3D, ou o ponto de uma superfície apontada por um cursor.
- **Seleção ou identificação.** Utilizada para fornecer à aplicação o identificador de um objeto gráfico de modo que ele possa ser distinguido de outros.
- **Entrada de texto.** Utilizada para a entrada de uma seqüência de caracteres.
- **Entrada de valor numérico.** Utilizada para a especificação de uma quantidade.
- **Entrada em uma opção.** Utilizada para adquirir uma alternativa dentre um conjunto de possibilidades pré-estabelecidas.

A partir destas tarefas básicas é possível definir as chamadas *tarefas de interação compostas*, tais como:

- **Caixas de diálogo.** Utilizadas para informar à aplicação um conjunto de informações não mutuamente exclusivas.
- **Construção.** Seqüência de posicionamentos utilizada para criar um objeto gráfico.
- **Manipulação.** Modificação de atributos de um objeto gráfico através de uma seqüência de combinações entre identificação, posicionamento ou entrada de um valor e definição de uma alternativa de ações.

De modo a aumentar a precisão nas tarefas de construção e manipulação, as aplicações freqüentemente impõem restrições sobre a execução das tarefas de interação, especialmente sobre as tarefas relacionadas ao posicionamento, segundo a escolha do usuário:

- **Alinhadores contínuos**. Restrição que consiste em limitar a tarefa de posicionamento sobre um objeto gráfico, restringindo as posições a linhas e planos previamente definidos. Bier [1990] chamou tal restrição de *alinhador*. Mesmo que restrito a esses elementos, o posicionamento continua sendo realizado em um espaço contínuo, porém com menor número de graus de liberdade.

- **Alinhadores discretos**. Consiste em fazer com que a tarefa de posicionamento seja realizada de forma discreta sobre um objeto gráfico. Comumente o alinhamento discreto restringe o posicionamento a pontos igualmente espaçados sobre uma reta ou nós de um reticulado (grade).

- **Campo de atração**. Tipo de restrição que define, para um objeto gráfico, uma região de influência de atração do cursor de um dispositivo apontador. O cursor, ao entrar nesta região, é deslocado (imediatamente ou suavemente) ao ponto mais próximo do objeto gráfico. Essa restrição simula um campo de gravidade que atrai o cursor ao objeto quando o cursor se aproxima do mesmo. Bier [1990] usou o termo *fixar* (*snap*) para designar esta atração.

Nesta tese utilizaremos o termo *restrição* para designar tanto a restrição de *fixar* como a de *alinhar*. Em todos os casos, explicitaremos o tipo de objeto gráfico sobre o qual a restrição de posicionamento é realizada (*e.g.*, posicionamento restrito a superfícies, restrito a linhas ou restrito a pontos). Na literatura, esse procedimento é comumente chamado de *snapping*.

2.2 Manipulação direta

Manipulação direta é o nome dado ao estilo de interação que permite ao usuário sentir como se ele estivesse controlando diretamente os objetos de interesse [Shneiderman, 1983]. As ações sugerem procedimentos físicos (*e.g.*, apontar, pegar, arrastar, soltar) e objetos são representados através de metáforas do mundo real. Nesse estilo de interação, o uso de realimentação visual eficiente e incremental permite que o usuário observe imediatamente o resultado de suas ações sobre os objetos. Usualmente, as ações são reversíveis e o objeto que sofre a ação está sempre visível.

Embora o termo *manipulação direta* tenha sido introduzido em 1982, esse estilo de interação tem sido utilizado em Computação Gráfica desde a década de 1960. O sistema Sketchpad permitia que o usuário utilizasse uma caneta óptica para apontar e selecionar objetos gráficos 2D na tela [Sutherland, 1963]. Este sistema também implementava uma função de campo de atração de modo a atrair o cursor

da caneta óptica a pontos, curvas ou interseções entre curvas, ajudando assim o usuário a posicionar e criar objetos com maior precisão. Tarefas básicas como posicionar e selecionar objetos gráficos utilizando um dispositivo apontador, e o uso de funções de restrição do posicionamento do cursor de acordo com a geometria manipulada, tornaram-se difundidas entre aplicações de desenho por computador e hoje são amplamente utilizadas em sistemas gráficos e jogos.

A manipulação direta através de dispositivos apontadores 2D é hoje o método de interação mais utilizado em sistemas de janelas. Em particular, desde a introdução em 1984 do sistema de janelas System 1 nos computadores Macintosh, X Windows System no sistema operacional UNIX, e a posterior popularização da interface gráfica do Microsoft Windows, o uso de um *mouse* como dispositivo apontador para selecionar e arrastar elementos gráficos tornou-se padrão em praticamente todas as aplicações nesses sistemas.

Jogos de *video game* também são exemplos evidentes do uso de manipulação direta em uma interface gráfica. Com pouco treinamento, os usuários são capazes de utilizar controles (*joysticks*) que mapeiam as ações físicas que o usuário impõe sobre botões e alavancas para ações correspondentes da personagem na tela. Entretanto, o uso do dispositivo apontador é mais comumente empregado em jogos de computador, seja para selecionar e arrastar elementos gráficos, seja para definir a orientação da personagem ou câmera que está sendo controlada.

Tradicionalmente, as tarefas de manipulação direta são feitas através de algoritmos executados na CPU, como tratadores de eventos disparados por um sistema de janelas. Em particular, a tarefa de seleção usando um dispositivo apontador 2D é reduzida à determinação de todos os objetos do banco de dados da cena que geram interseções com um raio que passa por dois pontos: a posição do cursor captada pelo sistema de janela e a posição da câmera definida pela aplicação. A figura 2.1 ilustra este procedimento. Essa técnica, conhecida como *ray picking*, é a mais utilizada pelas atuais bibliotecas gráficas 3D. No restante desta seção, sintetizamos a evolução destas bibliotecas, com o intuito de mostrar que as capacidades das GPUs têm sido pouco consideradas. Em todas as bibliotecas analisadas, as operações de interação são realizadas na CPU através do acesso aos modelos geométricos em um banco de dados de cena. Isto tem gerado um distanciamento, cada vez maior, entre o que se pode visualizar e o que tais bibliotecas podem oferecer em relação ao desenvolvimento de aplicativos com manipulações diretas.

Na década de 1980 surgiram as primeiras bibliotecas gráficas 3D baseadas em uma organização da cena através de estruturas de dados hierárquicas para facilitar o desenvolvimento de aplicativos interativos 3D. Nessa época, a ACM SIGGRAPH (*Special Interest Group on Graphics*, da *Association for Computing Machinery* [ACM, 2007]) havia proposto um padrão de APIs de gráficos independentes do dispositivo, chamado de *3D Core Graphics System* (ou simplesmente *Core*) [Committee, 1977]. Com base nesse padrão, foi desenvolvida em 1985 a biblioteca GKS (*Graphics Kernel System*) com

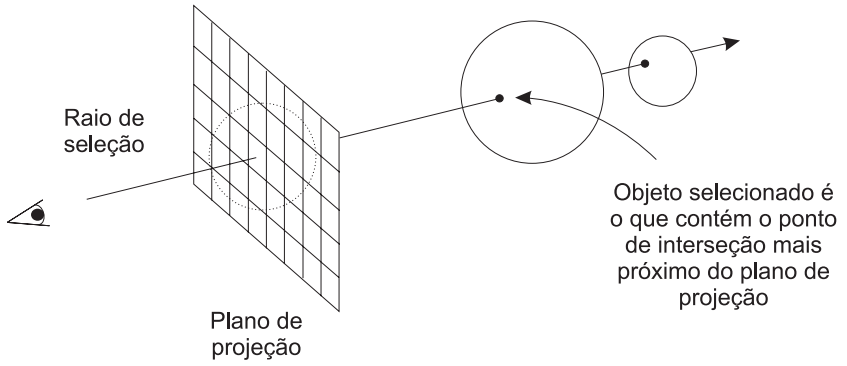


Fig. 2.1: Seleção usando *ray picking*.

funções de suporte a desenho de primitivas 2D [ISO, 1985]. Uma extensão para gráficos 3D foi proposta em 1988, porém ainda sem contemplar o agrupamento de primitivas em relações hierárquicas, e sem suporte direto a tarefas de manipulação direta.

A possibilidade de agrupar primitivas geométricas através de relações de hierarquia surgiu com o lançamento da API PHIGS (*Programmer's Hierarchical Interactive Graphics System*) em 1988 [van Dam, 1988, Gaskings, 1992]. Conjuntamente com o lançamento da API OpenGL em 1992, que sobrepôs o *Core* e PHIGS e tornou-se a API padrão de indústria para gráficos 2D e 3D independentes do dispositivo, surgiram as primeiras bibliotecas gráficas 3D baseadas em arquiteturas de interface gráfica que fazem o uso de grafos de cenas e ao mesmo tempo fornecem funcionalidades de interação por manipulação direta. Essas propostas são baseadas no paradigma da Arquitetura Gráfica Unificada (*Unified Graphics Architecture*) de Zeleznik et al. [1991], no qual os dados necessários para realizar interações, tais como a restrição do posicionamento de um cursor sobre uma superfície de uma geometria arbitrária, são calculados com relação aos dados de modelos geométricos representados por expressões analíticas e armazenados na memória do sistema.

As bibliotecas de grafo de cena são comumente chamadas de bibliotecas *retained mode*, em oposição às bibliotecas de comandos de renderização controladas diretamente pelo programador, chamadas de bibliotecas *immediate mode* (*e.g.*, OpenGL e Direct3D). Bibliotecas *immediate mode* não são capazes de manter uma representação persistente dos objetos gráficos da cena. Para realizar isso, bibliotecas *retained mode* usam estruturas de dados capazes de armazenar a representação de cada modelo, bem como suas relações com outros modelos e sua forma de apresentação ao usuário. A motivação para o desenvolvimento de bibliotecas *retained mode* está na necessidade de reduzir o tempo de desenvolvimento de aplicações que envolvem visualização de gráficos 3D e interação. De forma transparente ao desenvolvedor, tais bibliotecas utilizam técnicas de visibilidade e de gerenciamento de recursos para melhorar a eficiência da visualização de cenas complexas ou com grande

quantidade de oclusão de visibilidade.

2.2.1 Open Inventor e OpenGL Performer

A mais popular das bibliotecas gráficas baseadas em grafo de cena é o Open Inventor [Wernecke, 1994], produzida pela mesma desenvolvedora do OpenGL, a SGI - Silicon Graphics, Inc. [SGI, 2007b], e derivada do IRIS Inventor [Strauss, 1993]. Sua introdução no mercado teve como objetivo estimular o desenvolvimento de novas aplicações baseadas em OpenGL, uma vez que muitas aplicações deixavam de utilizar o OpenGL em razão da dificuldade de aprendizado para desenvolvedores pouco familiarizados com teoria de Computação Gráfica 3D. Com o Open Inventor, aplicações de gráficos 3D com interação poderiam ser criadas com mais facilidade e em menor tempo de desenvolvimento.

O Open Inventor é constituído de uma biblioteca em C++ e fornece uma camada de abstração de alto nível para visualização e interação usando OpenGL. Novos objetos são criados a partir de objetos mais simples (*e.g*, caixas, esferas, cilindros) definidos como nós de um grafo dirigido acíclico. Relações de hierarquia são utilizadas para encapsular objetos mais complexos e criar roteiros de animação. A determinação de visibilidade também é realizada internamente, e um conjunto de ferramentas de manipulação é disponibilizado ao desenvolvedor para criar funcionalidades de interação usando manipulação direta.

O Open Inventor está baseado fundamentalmente numa base de dados que contém uma representação da cena 3D (o *scene database*) como um grafo. Cada elemento (chamado de *node*) desse grafo representa um determinado objeto, que pode ser uma forma geométrica (*shape node*), um atributo da forma geométrica (*property node*) ou um nó que conecta outros nós em grafos e subgrafos (*group node*). Também são disponibilizados nós que representam câmeras e luzes. As ações realizadas sobre as formas geométricas são igualmente tratadas como objetos.

A aplicação de uma ação a uma determinada cena define um objeto que percorre, em profundidade, o grafo de cena. Esse novo objeto é utilizado para realizar operações tais como renderizar a cena, calcular o volume envoltório de um objeto ou grupo de objetos, realizar uma procura ou gravar a cena em arquivo. O comportamento pode ser especificado pelo próprio nó, impondo à ação um determinado modo de travessia do subgrafo subseqüente.

Determinados nós do *group node* funcionam como separadores (*separator nodes*), e têm a propriedade de gravar o estado atual do grafo, restaurando-o após a passagem da ação. A utilização dessa separação é desejável quando ocorrem várias instâncias de um mesmo objeto na mesma cena, impedindo que a alteração de um deles implique na alterações dos outros. O Open Inventor também permite reunir características de uma cena em um subgrafo chamado *node kit*, que pode ser utilizado e repetido arbitrariamente na cena principal.

Outra ferramenta disponível no Open Inventor é o *sensor*. Um sensor de dados (*data sensor*) pode ser utilizado para disparar um evento assim que houver uma mudança na base de dados da cena. Da mesma forma, um sensor de tempo (*timer sensor*), pode ser configurado para acionar um evento em intervalos regulares de tempo.

O Open Inventor conta com um mecanismo simples de tratamento de eventos. Um evento específico do sistema de janelas é gerado como resultado de alguma ação do usuário (movimentação do mouse, digitação, etc.) e é passado para a instância da área de renderização que corresponde à janela onde ocorreu o evento. Se o componente utilizado não processar o evento, ele é convertido para um evento independente do sistema de janelas e distribuído pelos nós, para o caso de algum deles tratá-lo de maneira específica.

A flexibilidade do Open Inventor está resumida na adição de novos *node kits* e na chamada de funções durante o percurso do grafo. Cada nó pode conter uma chamada de retorno (*callback node*) que invoca uma função configurável pela aplicação, quando o nó é percorrido, quando está a ponto de ser alcançado, ou quando acabou de ser alcançado. Isto permite introduzir comportamentos especializados dentro do grafo da cena e acessar o estado atual do percurso do grafo.

Os *widgets* de interação 3D do Open Inventor – chamados de *manipulators* – são criados através da composição de objetos geométricos simples chamados *draggers*. Essas composições definem objetos especiais capazes de tratar eventos do dispositivo apontador de modo a rodar, transladar e mudar a escala dos objetos da cena associados a cada *manipulator*.

Todas as propriedades de uma cena do Open Inventor, também armazenadas como nós do grafo, podem ser gravadas em um único arquivo em formato texto para posterior reutilização, possivelmente em diferentes aplicações. Essa e outras características do Open Inventor têm sido utilizadas até hoje em bibliotecas de visualização baseadas em grafo de cena, como o OpenSG [OpenSG, 2007] e OpenSceneGraph [OpenSceneGraph, 2007]. Essas propostas ainda são baseadas na Arquitetura Gráfica Unificada, centralizadas em torno de um banco de dados da cena na memória do sistema.

Desde o seu lançamento, a SGI observou que o Open Inventor atingia um desempenho muito inferior ao de aplicações criadas exclusivamente com o OpenGL, e portanto não era adequado para aplicações de alto desempenho. Essa preocupação motivou, nos meados da década de 1990, a criação de uma nova API: o OpenGL Performer [Eckel and Jones, 2004]. Em seu lançamento, o Performer tinha a capacidade de realizar processamento paralelo em sistemas com múltiplos processadores e possuía um grafo de cena mais flexível para se adaptar a diferentes necessidades de aplicações, ainda que sob o custo de reduzir as variedades de primitivas que poderiam ser utilizadas. Atualmente o Performer incorpora elementos de várias tecnologias desenvolvidas pela SGI [Sil, 2005], tais como o Volumizer (API de visualização de dados volumétricos), Vizserver (ferramenta para o uso de servidores de visualização), Multipipe SDK (API de escalabilidade de fluxos de renderização), Optimizer

(biblioteca de grafo de cena), além do Inventor propriamente dito. Por outro lado, todas as funcionalidades de interação continuam sendo processadas na CPU, desconsiderando quaisquer modificações da geometria ocorridas na GPU.

2.2.2 Cosmo3D

Em 1997, após o lançamento do OpenGL Performer, a SGI observou que o Open Inventor e Performer poderiam ser integrados em um único produto objetivando ao mesmo tempo facilidade de programação e alto desempenho. Em um trabalho conjunto com a Sun Microsystems [Sun, 2007a], começou então a ser desenvolvido o Cosmo3D [Eckel, 1998], uma biblioteca de grafo de cena baseada em VRML (*Virtual Reality Modeling Language*), com suporte a processamento paralelo e recursos de manipulação direta do Inventor, também limitadas a processamento na CPU.

Observando a evolução do conjunto de APIs do MS DirectX [Microsoft, 2007] nos PCs, a SGI desenvolveu o Cosmo3D de modo a ser compatível com plataformas Windows e não apenas com o tradicional sistema operacional IRIX [Graphics, 2007] da SGI. Mesmo assim, o Cosmo3D foi abandonado ainda em sua versão beta por diferentes motivos, incluindo problemas de projeto da arquitetura, o fato da Sun ter abandonado o projeto para concentrar seus esforços no desenvolvimento do Java3D [Sun, 2007b], e motivos estratégicos. Acima de tudo, a SGI queria produzir um produto que pudesse envolver mais indústrias da área, de modo a ter um padrão definitivo de bibliotecas gráficas *retained mode*, assim como já ocorria com o OpenGL como uma API *immediate mode*. Posteriormente, o Cosmo3D foi incorporado ao OpenGL Optimizer, um projeto proprietário da SGI e que atualmente não é mais suportado [SGI, 2007a].

2.2.3 OpenGL++ e Fahrenheit

O OpenGL++ [Rost, 1997] surgiu no início de 1997 como uma proposta de melhoramento do Cosmo3D, em um esforço conjunto entre a SGI, IBM [IBM, 2007], Intel Corporation [Intel, 2007] e a antiga Digital Equipment Corporation, hoje parte da Hewlett-Packard Development Company [HP, 2007]. Da mesma forma que o Inventor, o projeto do OpenGL++ era baseado em uma arquitetura de grafo de cena, com funcionalidades de determinação de visibilidade e manipulação direta. Incorporava características de multitarefa, modificação do grafo de cena em tempo real de modo a melhorar o desempenho, e possuía uma arquitetura mais bem projetada em relação ao Cosmo3D.

Tal como ocorreu com o Cosmo3D, o OpenGL++ jamais chegou a ser lançado, tendo sido abandonado por seus desenvolvedores. Em dezembro de 1997, o projeto deu lugar a uma nova proposta de API padrão de grafos de cena, desta vez numa parceria entre a SGI, a HP e a Microsoft: a API Fahrenheit [Insinger, 1998]. Nessa época, a API Direct3D conquistava um espaço cada vez mais significativo

na indústria de desenvolvimento de jogos para PCs, enquanto que o Open Inventor começava a perder espaço para a biblioteca Direct3D *Retained Mode*, também baseada em grafos de cena com funcionalidades de interação. Por um lado, o Fahrenheit serviria de salvaguarda à SGI, pois esta estava perdendo seu espaço no mercado ante a rápida popularização das aceleradores gráficas em PCs e a conseqüente expansão do DirectX. Por outro lado, a Microsoft via no Fahrenheit uma oportunidade de não deixar que o OpenGL++ se tornasse padrão de indústria sem a sua participação, e pretendia integrar (e posteriormente substituir) o próprio Direct3D ao (pelo) Fahrenheit.

Após dois anos de desenvolvimento, o projeto Fahrenheit foi abandonado por dificuldades de gerenciamento das indústrias envolvidas. A SGI começou a restringir continuamente o desenvolvimento do projeto de modo a tornar mais conveniente a compatibilidade do Fahrenheit com ferramentas proprietárias já existentes no sistema IRIX. Isso facilitaria a transição da SGI do seu mercado de estações gráficas dedicadas para o mercado de PCs equipados com aceleradoras gráficas. Como resposta, a Microsoft reduziu significativamente o tamanho da equipe de desenvolvimento, sinalizando que não pretendia mais transformar o Direct3D no Fahrenheit.

Com o fracasso do Fahrenheit, não foram mais propostas padrões de indústria para APIs baseadas em grafos de cena. Entretanto, as alternativas mais populares atualmente, tais como o OpenSG e OpenSceneGraph, são bibliotecas de grafo de cena com licença de código aberto, baseadas em APIs *immediate mode* como o OpenGL ou Direct3D. Assim como o Open Inventor, essas APIs possuem suporte a *widgets* de interação, seja de forma integrada ou através de extensões. Porém, tais funcionalidades de interação são incapazes de considerar possíveis modificações de geometria realizadas em *hardware* gráfico. Todo o processamento é realizado exclusivamente na CPU, gerando inconsistências entre a visualização do modelo deformado na GPU e a interação com esse modelo.

2.3 Evolução do *hardware* gráfico

Nos últimos anos, a comunidade de Computação Gráfica presenciou avanços extraordinários no uso do *hardware* gráfico em computadores pessoais. Popularizada especialmente pela indústria de jogos, as GPUs evoluíram não só com relação ao desempenho, mas também, e principalmente, em novas funcionalidades implementadas, das quais destaca-se a capacidade de programação dos processadores gráficos. Em paralelo, constatou-se uma redução substancial do custo desses equipamentos. Técnicas de renderização que, há uma década, só poderiam ser executadas em tempo real em estações gráficas de centenas de milhares de dólares, hoje são facilmente encontradas em jogos que utilizam placas gráficas populares de custo inferior a uma centena de dólares. Essa rápida evolução deve continuar nos próximos anos. Atualmente, o desempenho das GPUs tem crescido numa taxa que excede a taxa de crescimento de desempenho observada nos microprocessadores de propósito geral: 225% ao

ano para as GPUs [Fernando, 2004], contra um máximo de 50% do crescimento anual do desempenho das CPUs [Dally, 1999]. Contudo, tal evolução tem sido concentrada no melhoramento de técnicas de síntese de imagens em oposição ao melhoramento de técnicas de interação 3D. Do nosso conhecimento, o máximo que se tem feito para se adequar às deformações geométricas ocorridas na GPU é implementar uma cópia das funções de deformação na CPU e utilizá-las sempre que uma interação com o modelo for necessária. Neste trabalho, apresentamos um paradigma inovador que, tirando as vantagens de processamento da GPU, consegue identificar os pontos de interseção entre um raio de seleção e os objetos da cena sem recorrer à replicação das funções de deformação na CPU.

Para contextualizar melhor a nossa contribuição, apresentamos nesta seção um histórico da evolução do *hardware* voltado a aceleração de técnicas de visualização, de modo a contrapor com a evolução das bibliotecas de interação por manipulação direta citadas anteriormente. Em especial, concentramo-nos na evolução dos sistemas compatíveis com as APIs gráficas GL. Isso inclui todos os sistemas produzidos pela SGI com fluxo de renderização compatível com APIs como OpenGL [Shreiner et al., 2005] e Direct3D [Microsoft, 2006]. Inclui, também, todas as placas gráficas utilizadas atualmente em computadores pessoais, e os sistemas compatíveis com as já obsoletas APIs IrisGL [McLendon, 1992] da SGI e Glide [3Dfx, 1997] da então 3Dfx Interactive, Inc. Essa linha de evolução incorpora mais de 20 anos de desenvolvimento de *hardware* gráfico: de 1983 à 2007. Em particular, dividimos essa cronologia em quatro gerações de acordo com o conjunto de funcionalidades implementadas em comum [Akeley and Hanrahan, 1999]. Historicamente, tais funcionalidades têm sido definidas conjuntamente pelas indústrias de *hardware*, desenvolvedores de APIs gráficas e desenvolvedores de aplicações, levando em conta a análise de custo e demanda do mercado. Indiretamente, essa evolução reflete também o aprimoramento das GPUs em termos de desempenho e qualidade da imagem gerada. Em especial, as quatro gerações são as seguintes:

- **Primeira geração.** Constituída de *hardware* gráfico capaz de realizar apenas transformações geométricas em vértices e recorte de polígonos.

- **Segunda geração.** Incorpora as características da primeira geração, e inclui a capacidade de processar uma equação pré-definida de iluminação para cada vértice.

- **Terceira geração.** Incorpora as características das gerações anteriores, e acrescenta funcionalidades de amostragem de texturas.

- **Quarta geração.** Inaugura a capacidade de programação dos processadores de vértices e fragmentos através da API gráfica.

2.3.1 Primeira geração: transformação geométrica

A primeira geração de *hardware* gráfico compatível com APIs gráficas GL compreende a família de sistemas gráficos da SGI produzidos no período de 1983 à 1987. Essa categoria inclui as primeiras estações e terminais lançados pela SGI, contendo então os primeiros sistemas gráficos produzidos em série, como os terminais da série IRIS (*Integrated Raster Imaging System*) 1000 lançados em 1983. As séries IRIS 2000 e 3000 também são incluídas nesta classe. Um sistema IRIS 2000 (1984) era capaz de processar cerca de 30.000 vértices por segundo e preencher 46 milhões de *pixels* por segundo.

Inicialmente essa família era constituída de *hardware* capaz de exibir apenas modelos aramados (*wireframe*), e só posteriormente incluiu os primeiros sistemas capazes de renderizar triângulos preenchidos com cores sólidas. O mecanismo de rasterização era capaz de desenhar pontos e linhas com interpolação de cores, e triângulos preenchidos com uma cor sólida. O *buffer* de profundidade era inexistente.

O *hardware* gráfico desses primeiros sistemas gráficos da SGI já era equipado com um fluxo completo de processamento de vértices, incluindo transformação geométrica, recorte com relação ao volume de visualização e projeção. Funcionalidades similares só seriam implementadas em *hardware* gráfico de computadores pessoais a partir de 1999 com o surgimento da placa gráfica NVIDIA GeForce 256.

2.3.2 Segunda geração: triângulos sombreados e iluminação

Os sistemas SGI GT da série *Professional IRIS Series*, modelos 4D(50-85), lançados a partir de 1987, inauguraram a segunda geração de *hardware* gráfico, caracterizada pela introdução da rasterização de triângulos preenchidos com interpolação de cores segundo o modelo *Gouraud* [Gouraud, 1971], suporte a *buffer* de profundidade e capacidade de combinar cores de acordo com o valor de opacidade de cor nos vértices (*alpha blending*). Tão importante quanto essas características, tais sistemas eram capazes de realizar todo o processamento de iluminação de vértices, *i.e.*, podiam avaliar a equação do modelo de iluminação para obter a contribuição de luz difusa e especular para cada vértice, sem depender da CPU. Da mesma forma que o processamento de transformação geométrica, essa característica só surgiu em *hardware* gráfico para computadores pessoais a partir de 1999. De acordo com Corda [1996], o sistema gráfico GTX dessa segunda geração podia trabalhar com uma resolução de até 1280×1024 *pixels*, processar mais de 405.000 vértices por segundo e 80 milhões de *pixels* por segundo.

2.3.3 Terceira geração: texturização

Em 1992, é lançado pela SGI o OpenGL: uma especificação de interface de *software* com o *hardware* gráfico. Construída como um aprimoramento do IrisGL (API utilizada nos sistemas SGI de gerações anteriores), o OpenGL tornou-se a API padrão de indústria desde então, tendo sido mantida e atualizada por um comitê independente de indústrias de *hardware* e *software* da área: o comitê ARB - *Architecture Review Board*. Em setembro de 2006 o comitê ARB foi dissolvido, e o controle do OpenGL passou a ser realizado por um grupo de trabalho da Khronos [Khronos, 2007] - um comitê formado por mais de 100 indústrias relacionadas a atividades de autoria e aceleração de mídia, com o objetivo de definir APIs padrões de indústria e de uso livre.

No início da década de 1990, as estações de alto desempenho suportavam apenas um número reduzido de características do OpenGL, sendo o restante simulado em *software* pela própria API. O sistema *RealityEngine* [Akeley, 1993], lançado em 1992 com um custo aproximado de 1 milhão de dólares, marcou o início dessa terceira geração como o primeiro *hardware* gráfico capaz de fornecer aceleração de *hardware* para todas as etapas de processamento de renderização especificadas pelo OpenGL 1.0, em especial o suporte a texturização usando texturas 2D com *mip-mapping* e suporte a texturas 3D. Em sua primeira versão, o *RealityEngine* era capaz de processar cerca de 6 milhões de vértices por segundo e 380 milhões de *pixels* por segundo.

A terceira geração de *hardware* gráfico perdurou até 2000 quando surgiram os processadores gráficos programáveis em computadores pessoais. Antes disso, porém, no período relativamente curto de 1995 à 2000, as placas gráficas de computadores pessoais evoluíram de GPUs inferiores aos sistemas SGI de primeira geração para alcançarem o mesmo nível de desempenho e características disponíveis anteriormente apenas em estações SGI de alto desempenho e custo elevado. Além disso, o surgimento de novas funcionalidades em *hardware* gráfico para PCs ocorreu em ordem diversa daquela observada nas estações SGI. Enquanto os primeiros sistemas SGI já eram equipados com fluxos completos de transformação de vértices, porém sem qualquer suporte à texturização, as antigas GPUs para PCs, produzidas simultaneamente pela S3 Graphics Co., Ltd. [S3, 2007], Matrox Graphics, Inc. [Matrox, 2007] e NVIDIA Corporation [NVIDIA, 2007b], eram equipadas com funcionalidades de texturização, porém sem transformação de geometria ou processamento de iluminação de vértices. Eram capazes de realizar apenas varredura de linhas não texturizadas e, em alguns casos, tinham desempenho até mesmo inferior ao código de máquina otimizado na CPU. Além disso, sofriam seriamente de falta de suporte adequado de APIs. Nesse período, a arquitetura de tais placas era incompatível com aquela sobre a qual o OpenGL foi projetado.

As GPUs, que ganharam popularidade no mercado de PCs, foram as placas da série *Voodoo*, lançadas a partir de 1996 pela 3Dfx. As primeiras placas *Voodoo* eram capazes de exibir triângulos texturizados com *mip-mapping* e filtragem bilinear. Entretanto, o *hardware* ainda dependia da

CPU para preparar os triângulos para a rasterização. As primitivas só poderiam ser processadas pelo *hardware* gráfico se fossem previamente convertidas em trapézios degenerados, alinhados em coordenadas da tela. Tal processo deveria ser feito em *software*, o que prejudicava o desempenho geral de renderização. Com exceção do fluxo de transformação e iluminação de geometria, a placa possuía funcionalidades de rasterização equivalentes a um sistema SGI *RealityEngine*. A 3Dfx tornou-se rapidamente líder na indústria de *hardware* gráfico de PCs graças a sua estratégia de fornecer uma API poderosa e ao mesmo tempo bem documentada (porém proprietária) para trabalhar em conjunto com suas placas: a API *Glide* [3Dfx, 1997]. As placas 3Dfx predominaram até 1999 quando então o OpenGL e a API Direct3D se tornaram as APIs definitivas para as placas utilizadas por concorrentes importantes como a ATI Technologies, Inc. [ATI, 2007a], Matrox e NVIDIA.

O desenvolvimento das placas gráficas para PCs segue a evolução da API Direct3D, da Microsoft. Em 1995, a Microsoft lança o *Windows 95 Games SDK*, um conjunto de APIs de baixo nível para o desenvolvimento de jogos e aplicações multimídia de alto desempenho. Em 1996, seu nome é modificado para DirectX e sua segunda e terceira versões são disponibilizadas em junho e setembro desse mesmo ano. Entre as APIs contidas no DirectX, o Direct3D é concebido como a API para uso de aceleradores gráficos 3D utilizando o mesmo fluxo de renderização especificado no OpenGL. Embora naquela época a API da Microsoft fosse duramente criticada por sua arquitetura demasiadamente confusa e mal projetada[1], ela começa a se tornar a API mais utilizada para desenvolvimento de jogos, uma vez que novas versões começam a ser divulgadas em intervalos mais curtos que aqueles do OpenGL, cujos aprimoramentos dependem da adição de novas extensões deliberadas pelo comitê ARB. Cada nova versão do Direct3D experimenta modificações profundas de projeto e vários aprimoramentos.

Em 1997 é lançado o DirectX 5 (por motivos comerciais, o DirectX 4 nunca foi disponibilizado e tornou-se o próprio DirectX 5), acompanhando as primeiras placas capazes de renderizar triângulos em geral, tais como a NVIDIA Riva 128 e a ATI Rage Pro. A Riva 128 não atinge a mesma qualidade de imagem produzida pelas placas 3Dfx, mas ultrapassa as placas *Voodoo* em várias medições de desempenho. Ainda assim, o fluxo de processamento de geometria é inexistente e a CPU é responsável por calcular todas as transformações geométricas e interpolações dos atributos de vértices ao longo das arestas para cada triângulo transformado. Em outras palavras, a CPU é responsável por interpolar as coordenadas X, Y e Z (em coordenadas da tela), a componente RHW (*Reciprocal Homogeneous W*: valor recíproco da coordenada homogênea, utilizada na correção de perspectiva durante a texturização), as coordenadas de textura, as componentes de cor especular, de cor da neblina e valores de

[1] Em 1996, John Carmack, desenvolvedor do jogo *Doom* (*id Software*), criticou publicamente a arquitetura do Direct3D e sugeriu que a Microsoft apoiasse o OpenGL. De forma semelhante, um artigo da revista *Game Developer Magazine*, de abril-maio de 1997 e assinada por Chris Hecker, analisava as duas APIs e concluía que a Microsoft deveria abandonar o Direct3D e usar o OpenGL [Hecker, 1997].

opacidade para todas as três arestas de cada triângulo.

Em 1998 é lançado o DirectX 6 juntamente com a primeira placa para PC capaz de calcular as interpolações de atributos ao longo das arestas. A CPU ainda é responsável pela transformação e iluminação de cada vértice, mas torna-se necessário fornecer apenas os atributos para cada vértice em vez de atributos interpolados para cada aresta de cada triângulo. Um ano depois, o DirectX 7 é lançado com vários aprimoramentos de projeto e com uma significativa facilidade de uso, imitando as características do OpenGL. Placas gráficas equipadas com múltiplos fluxos de texturização, tais como as placas NVIDIA TNT e TNT2, tornam-se rapidamente difundidas.

Em 2000, as placas populares evoluem para as primeiras placas equipadas com uma arquitetura capaz de implementar o fluxo completo de transformação e iluminação de vértices, com centenas de estágios de processamento paralelo. Em setembro daquele ano, o lançamento do DirectX 8 fornece suporte ao desenvolvimento de aplicações capazes de explorar todas essas características. As placas gráficas para computadores pessoais finalmente ultrapassam as características do sistema SGI *RealityEngine*, mas ao mesmo tempo obtendo uma redução de custo de mais de 99.99% em comparação com o antigo sistema SGI: de um sistema de 1 milhão de dólares, do sistema *RealityEngine*, para uma placa de custo inferior a uma centena de dólares. De acordo com a "Lei de Moore"[2], e observando a diminuição do custo das CPUs nesse período, era estimado que tais placas custassem muito mais, em torno de 15 mil dólares. Ainda neste momento, porém, não existia a possibilidade de realizar processamento programável de vértices na GPU. Assim, o processamento para interação poderia ser realizado na CPU sem qualquer tipo de desvantagem.

2.3.4 Quarta geração: processadores programáveis

O maior avanço introduzido na última geração de *hardware* gráfico, e principal fator responsável pela flexibilidade de uso desse equipamento em aplicações de Computação Gráfica, é o advento de GPUs programáveis. Até então, o fluxo de renderização segundo o OpenGL ou Direct3D poderia ser configurado apenas através de um conjunto fixo de estados. A aplicação poderia escolher entre modelos de interpolação utilizados na rasterização (*e.g.*, modelo *flat* ou *smooth* segundo o OpenGL), alterar as componentes de reflexão difusa, ambiente, especular e de emissão das superfícies segundo o modelo de iluminação da API gráfica GL (baseado no modelo de Bui-Tong [1975]), selecionar e configurar alguns tipos de fontes de luz (*e.g.*, ambiente, direcional, pontual ou *spot*), e modificar um conjunto de variáveis de controle de combinação de texturas. O uso desses atributos durante a

[2]A Lei de Moore deve-se a uma afirmação de Gordon Moore, ex-presidente e atual *Chairman Emeritus* da Intel Corp. que, em 1965, observou que a capacidade de processamento dos microprocessadores de computadores pessoais dobrava a cada 18 meses como reflexo do aumento no número de transistores na mesma proporção. Esse comportamento é observado até hoje.

etapa de transformação geométrica e iluminação era pré-determinado e à aplicação só cabia habilitar/desabilitar estados desse fluxo.

Com as GPUs programáveis, a aplicação passa a ser capaz de redefinir o comportamento das etapas de transformação geométrica e iluminação, tanto com relação aos vértices como aos fragmentos, através da programação de um processador de vértices e um processador de fragmentos. O processador de vértices é responsável pela execução de um programa que modifica os atributos de cada vértice da geometria submetida pela CPU. Por sua vez, o processador de fragmentos é responsável pela execução de um programa que processa cada fragmento de *pixel* gerado durante a etapa de rasterização e assim determina atributos de cor e profundidade a serem direcionados a *buffers* de renderização, também chamados de *alvos de renderização* (*render targets*). As placas gráficas mais recentes contam ainda com uma arquitetura unificada, capaz de trabalhar tanto como um processador de vértices, de fragmentos, ou um processador de primitivas, de acordo com a demanda. Este último processador tem a capacidade de acessar as informações associadas a cada primitiva, como os vértices que compõem a primitiva, além das relações de conectividade com as primitivas adjacentes.

Ao explorar a flexibilidade do processador de vértices, fragmentos e geometria, a aplicação tem a oportunidade de implementar em *hardware* um número variado de modelos de iluminação, incluindo os clássicos modelos de Phong [Bui-Tong, 1975] e Blinn [Blinn, 1977] (para uma implementação na GPU, veja Fosner [2002]), e também modelos mais complexos como os de Cook-Torrance [Cook and Torrance, 1982] e Oren-Nayar [Oren and Nayar, 1992] (veja também Engel [2004]), e modelo de Kubelka-Munk [Wendlandt and Hecht, 1966, Kortun, 1969] (implementação na GPU segundo Baxter et al. [2004]). Essa flexibilidade se estende também à possibilidade de simular modelos de iluminação baseados em BRDF (*Bidirectional Reflectance Distribution Function*), calcular termos de reflexão de Fresnel (razão entre a quantidade de luz refletida e luz transmitida numa superfície), implementar modelos de iluminação anisotrópica e modelos de iluminação para ilustrações (modelos não fotorealistas), entre uma pletora de outras possibilidades.

A deformação da geometria através do processador de vértices inclui a habilidade de realizar *skinning* de vértices em *hardware* [Dempski, 2002] e mapeamento de deslocamento (*displacement mapping*) usando texturas convencionais ou texturas procedurais criadas pelo processador de fragmentos [Green, 2002]. No processador de fragmentos, os atributos de cor e profundidade dos *pixels* podem ser modificados segundo diferentes tipos de mapeamento de texturas para simulação de detalhes 3D [Policarpo et al., 2005, Krishnamurthy and Levoy, 1996, Tatarchuk, 2006].

Essa geração de *hardware* gráfico programável tem início em 2000 com o lançamento da placa gráfica NVIDIA GeForce 3 [Lindholm et al., 2001] para o computador Apple Macintosh. No início de 2001, durante o evento *MacWorld Expo Tokyo* sobre computadores Macintosh, é exibido o filme de curta metragem Luxo Jr., produzido originalmente pela Pixar [Pixar, 2007], mas desta vez sendo

renderizado em tempo real numa GeForce 3. Steve Jobs, então CEO (*Chief Executive Officer*) da Apple Computer, Inc. [Apple, 2007], ao se pronunciar a respeito dessa demonstração, lembra que "O que, há 15 anos, levava 55 horas para renderizar cada segundo de animação, é agora feito em tempo real." Mais tarde, as potencialidades de uma GPU programável similar, a NVIDIA Quadro DCC, são exibidas durante uma demonstração de tecnologia na conferência SIGGRAPH 2001. Em particular, é exibida uma versão interativa do filme *Final Fantasy*, da Square Enix Co., Ltd., também em tempo real. De acordo com informações do fabricante do *hardware*, cerca de 1,5 milhões de vértices são processados a cada segundo para renderizar apenas os cabelos de um dos modelos humanos mostrados nessa demonstração. O fabricante também destaca que o desempenho das operações em números de ponto flutuante utilizados para desenhar apenas um quadro deste filme é superior ao poder computacional total de um supercomputador *Cray* naquele momento. Por outro lado, a exibição dos efeitos utilizados nesses filmes não teria sido possível sem a capacidade de programação de vértices e fragmentos.

Os programas executados pelos processadores das placas gráficas são comumente chamados de *shaders*. Este termo foi introduzido por Cook [1984] para designar um conjunto de procedimentos de modificações de propriedades de geometria e fragmentos para simulação de diferentes materiais em síntese de imagens, e foi difundido especialmente no RenderMan [Hanrahan and Lawson, 1990]. O RenderMan é uma especificação técnica de interface entre aplicativos de modelagem geométrica e renderização fotorealista, capaz de descrever completamente uma cena 3D, incluindo a posição das fontes de luz, câmeras, geometria dos objetos e procedimentos de sombreamento e iluminação definidos pela aplicação. Tal especificação permite que uma implementação particular do RenderMan possa renderizar uma geometria de forma consistente de acordo com as propriedades dos materiais descritas pelo artista gráfico.

A interpretação dos *shaders* do RenderMan como programas modificadores de geometria e fragmentos é estendida naturalmente aos programas que podem ser executados nos processadores das GPUs de *hardware* gráfico de quarta geração. Assim, os nomes *vertex shader* e *fragment/pixel shader* são usualmente utilizados para designar os programas executados pelos processadores de vértices e fragmentos das atuais GPUs. De uma maneira muito parecida com a capacidade de programação do RenderMan, a aplicação pode implementar suas próprias equações de iluminação, procedimentos de transformação geométrica e descrição de materiais. De fato, um dos objetivos dessa nova geração de *hardware* gráfico é atingir a mesma flexibilidade dos *shaders* do RenderMan, porém com desempenho suficiente para aplicações em tempo real.

Além da programação de *shaders*, outras características ainda estão sendo desenvolvidas e aprimoradas nesta quarta geração de *hardware* gráfico. Estas incluem a manipulação de superfícies curvas (*Bézier* e NURBS), mapeamento de sombras em *hardware*, alvos de renderização com formato de

ponto-flutuante e precisão numérica capaz de armazenar valores de luminância com faixa dinâmica estendida e maior qualidade de filtragem de textura.

2.4 Arquitetura das GPUs programáveis

Nesta seção descrevemos os detalhes da arquitetura das GPUs programáveis sobre a qual a arquitetura de suporte a tarefas de interação 3D é construída. Também mostramos como as atuais GPUs podem ser utilizadas como processadores de fluxo de propósito geral, uma vez que esse tipo de processamento também é utilizado nos estágios de processamento de nossa arquitetura.

Em geral, as atuais GPUs são equipadas com dois processadores programáveis: o processador de vértices, segundo uma arquitetura MIMD (*Multiple Instruction Multiple Data*), e o processador de fragmentos, segundo uma arquitetura SIMD (*Single Instruction Multiple Data*). Recentemente surgiram placas gráficas equipadas com uma arquitetura SIMD unificada de multiprocessamento de primitivas, vértices e fragmentos, como as placas da série GeForce 8, da NVIDIA [NVIDIA, 2007a], e Radeon R600 [ATI, 2007b], da ATI. Entretanto, para melhor compreensão da evolução das placas gráficas programáveis, detalharemos neste texto apenas a arquitetura tradicional especificada pelas APIs gráficas GL, composta de um estágio de processamento de vértices e outro de processamento de fragmentos.

Os processadores de vértices e fragmentos não substituem completamente as funcionalidades do chamado *fluxo de função fixa* de renderização, que é o fluxo de renderização descrito pelo OpenGL para as gerações anteriores de *hardware* gráfico não programável. Em vez disso, o processamento realizado por esses processadores pode substituir apenas as tarefas de transformação geométrica, avaliação do modelo de iluminação para cada vértice, e texturização com definição de modos de combinação de texturas. Essa escolha de arquitetura tem sido adequada para manter a compatibilidade com aplicações gráficas até agora desenvolvidas para *hardware* gráfico de gerações anteriores e que utilizam APIs como OpenGL e Direct3D. Se a aplicação optar por não programar nenhum dos processadores, o *hardware* gráfico se comportará como um equipamento de terceira geração (ainda que, internamente, *shaders* sejam utilizados para simular o fluxo de função fixa).

Para propósitos de comparação, a figura 2.2 mostra um diagrama de blocos descrevendo o fluxo de função fixa de renderização utilizado em *hardware* gráfico a partir da terceira geração. O *hardware* recebe como entrada um conjunto de atributos de vértices e um conjunto de informações descrevendo que primitivas devem ser construídas com tais vértices. Esses dados passam inicialmente por um estágio de montagem de vértices (estágio 1 na figura 2.2), no qual os atributos de cada vértice são reunidos para processamento. Após esta etapa, cada vértice é processado por um fluxo de transformação geométrica (estágio 2). Em particular, a posição de cada vértice é modificada segundo uma

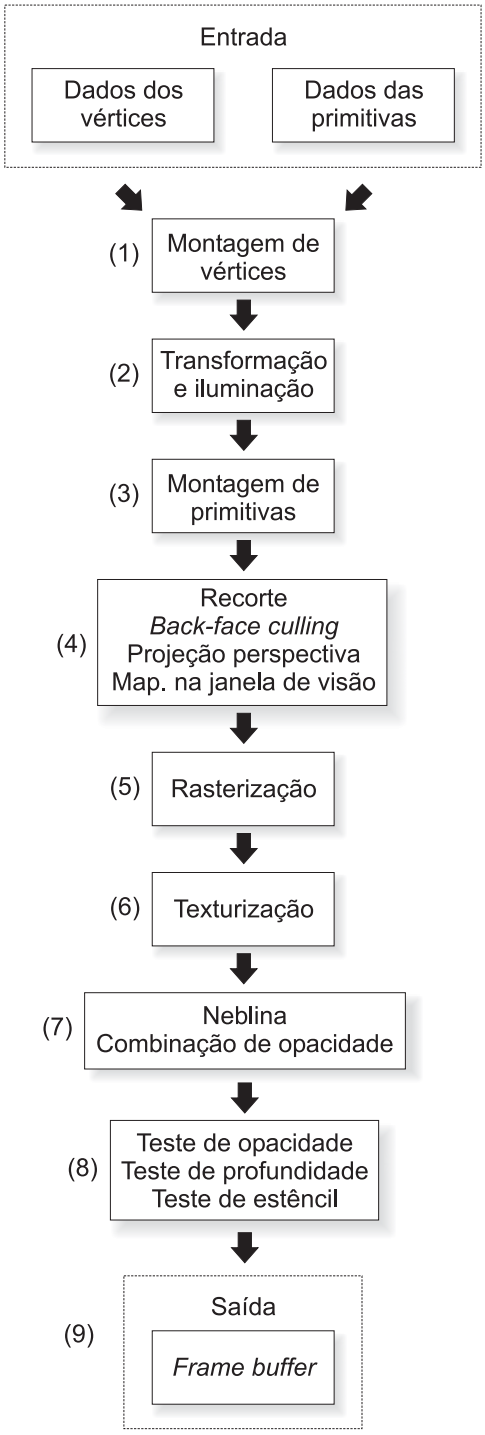


Fig. 2.2: Diagrama do fluxo de função fixa de renderização.

matriz de transformação do modelo (que converte coordenadas locais do modelo para coordenadas globais da cena), concatenada com a matriz de visão (que converte coordenadas globais para coordenadas relativas à câmera) e a matriz de projeção, resultando em um vértice descrito em coordenadas homogêneas do espaço de recorte do volume de visão canônico [Shreiner et al., 2005]. Nesse estágio, a fórmula do modelo de iluminação é processada para cada vértice e as informações de cores resultantes são incluídas em atributos de cores do vértice calculado. Na etapa seguinte (estágio 3), são montadas as primitivas a partir dos vértices processados (pontos, linhas e triângulos). Tais primitivas são recortadas pelo volume de visão e descartadas quando sua faces formam um ângulo maior do que 90 graus com relação ao vetor de visão (*back-face culling*). A posição de cada vértice é finalmente dividida por sua coordenada homogênea de modo a produzir o efeito de perspectiva e obter as coordenadas no sistema de coordenadas da janela de visão. Essa etapa (estágio 4) finaliza o processo de transformação geométrica e tem como resultado primitivas projetadas na tela, prontas para o processo de amostragem e processamento de fragmentos.

O rasterizador (estágio 5) gera os fragmentos dos *pixels* coincidentes com cada primitiva projetada na tela, obtendo atributos a partir dos vértices segundo o modelo de interpolação escolhido (*flat* ou *smooth*). Após serem texturizados segundo a amostragem e combinação de diferentes texturas (estágio 6), cada fragmento pode ser modificado de acordo com configurações de neblina (*fog*) e combinação de opacidade segundo uma componente de cor alfa (*alpha blending*) (estágio 7). Por fim, cada fragmento é testado segundo os testes de opacidade (*alpha test*), profundidade (*depth test*) e estêncil (*stencil test*) (estágio 8). Tendo passado por todos os testes, a cor resultante e o valor de profundidade do *pixel* são enviados ao *frame buffer* (estágio 9).

Uma vez que a atual arquitetura de *hardware* gráfico programável mantém a compatibilidade com o fluxo de função fixa, a aplicação pode escolher entre programar apenas o processador de vértices, apenas o processador de fragmentos, ou os dois ao mesmo tempo. Se apenas o processador de vértices é programado, o fluxo de renderização utilizará a função fixa de processamento de fragmentos. Da mesma forma, se apenas o processador de fragmentos for programado, o fluxo de função fixa de transformação e iluminação de vértices será utilizado. Por outro lado, não é possível utilizar o processador de vértices em conjunto com o fluxo de função fixa de processamento de vértices, ou o processador de fragmentos em conjunto com o fluxo de função fixa de processamento de fragmentos. Um diagrama em blocos ilustrando os estágios de processamento de um fluxo programável é mostrado na figura 2.3. As modificações são fundamentalmente a substituição da etapa de transformação e iluminação pelo processador de vértices, e substituição da etapa de texturização pelo processador de fragmentos.

No nível mais baixo de abstração, *shaders* de vértices e fragmentos são escritos através de linguagens semelhantes às linguagens de montagem das CPUs. Por outro lado, não são consideradas como linguagens de máquina, mas linguagens intermediárias cujas instruções são traduzidas em tempo de

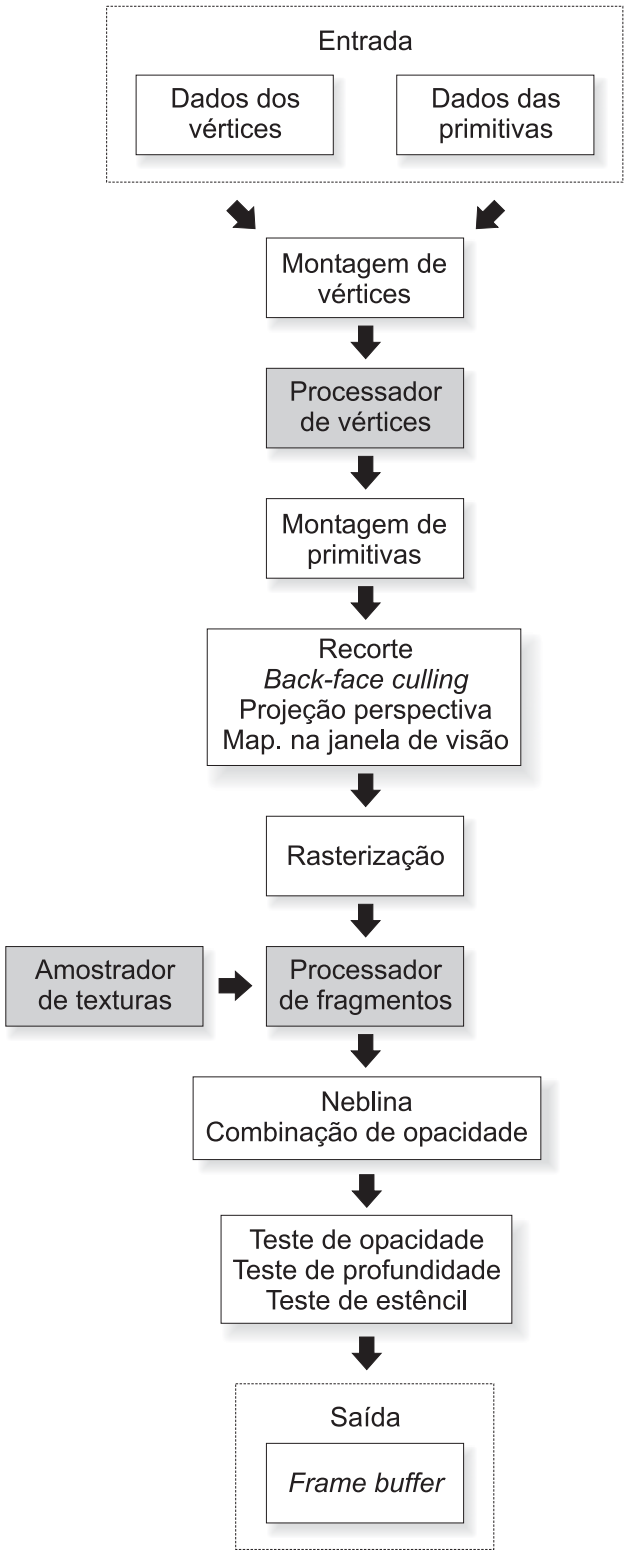


Fig. 2.3: Diagrama típico de um fluxo programável de renderização.

execução para a GPU através do *driver* de vídeo. Este é um processo semelhante ao que ocorre no modelo de compilação *Just-In-Time* utilizado em linguagens intermediárias de máquinas virtuais. Mas, contrastando ainda mais com programas tradicionais da CPU, *shaders* não são aplicações por si só, e sempre dependem de uma aplicação na CPU que utilize a API gráfica. Essa aplicação é responsável por configurar as funcionalidades não programáveis das GPUs, alimentar o fluxo de renderização com dados de primitivas, texturas e valores constantes, e definir os alvos de renderização. Em nossa proposta de uma arquitetura de suporte a tarefas de manipulação direta 3D usando a GPU, essa dependência exige que parte do processamento ainda seja realizada na CPU.

Diferentes versões das linguagens de *shaders* são disponibilizadas de acordo com o modelo do ambiente de execução de cada GPU programável. Tal modelo, chamado de modelo de *shader* (*Shader Model*), reflete o aprimoramento do conjunto de instruções e registradores implementados segundo as sub-gerações de *hardware* gráfico de quarta geração. O primeiro e mais limitado modelo de *shader* é o modelo 1.1, introduzido em 2000 com a placa NVIDIA GeForce 3. O modelo 3.0 é o modelo exigido na implementação da nossa arquitetura de interação. A arquitetura ilustrada na figura 2.3 é ligeiramente modificada quando se considera um *hardware* compatível com modelo de *shader* 3.0. A principal alteração é a capacidade do processador de vértices de também poder realizar amostragem de texturas. O modelo de *shader* 4.0 é o mais atual neste momento, e consta na especificação da API Direct3D 10. Sua principal modificação está na introdução de um processador de geometria capaz de acessar informações de adjacência dos vértices que compõem cada primitiva, além de permitir criar novas primitivas sem depender da CPU. Para uma comparação entre os diferentes modelos de *shaders*, veja Microsoft [2006] e Blythe [2006].

2.4.1 Processador de vértices

O processador de vértices é responsável pela execução de um *shader* para cada vértice da geometria obtido após a etapa de montagem de vértices. Um vértice é composto de até 16 atributos definidos pela aplicação segundo uma lista de formato de vértices pré-definida pela API. A aplicação também define a semântica de cada atributo segundo uma lista de significados disponibilizada pela API. A semântica dos atributos é utilizada na etapa de montagem dos vértices (*e.g.*, para distinguir qual atributo possui a posição do vértice) e para que o fluxo de função fixa de renderização possa distinguir, por exemplo, qual elemento corresponde ao vetor normal para o cálculo de iluminação, ou quais elementos contêm coordenadas de textura para a etapa de texturização. Essa lista de semânticas inclui a posição do vértice, vetor normal, vetor tangente e vetor binormal[3], cor $RGB\alpha$, coordenadas

[3]De acordo com Lengyel [2003], o termo *binormal* não é apropriado no contexto de mapeamento de detalhes 3D, pois sua definição em geometria diferencial não corresponde a um vetor tangente a um ponto da superfície, alinhado segundo a parametrização de uma das coordenadas de textura nessa superfície. O autor sugere o termo *vetor bitangente* [Lengyel,

de textura, entre outras.

A aplicação pode escolher os atributos de coordenadas de textura para armazenar dados arbitrários, *i.e.*, não necessariamente contendo coordenadas de textura, e interpretá-los no processador de vértices como valores numéricos genéricos que variam de acordo com o vértice. Entretanto, a saída do *shader* de vértices deve ter no mínimo um atributo com semântica de posição. Este deve conter a posição do vértice transformado, em coordenadas homogêneas do espaço de recorte. Os demais atributos são opcionais, e podem ser utilizados para propagar atributos de cor para o cálculo de iluminação do vértice e atributos de coordenadas de textura para o processador de fragmentos. Novos atributos também podem ser calculados e enviados à saída do *shader*, usando uma estrutura diferente daquela utilizada na entrada.

Um *shader* de vértices recebe como entrada apenas os atributos de um vértice de cada vez e retorna igualmente os atributos de um único vértice. Não é possível remover vértices de uma primitiva, nem adicionar novos vértices. O *shader* de vértices também não tem acesso às informações de conectividade do vértice que está sendo processado, a não ser que essa informação seja passada como um atributo definido pela aplicação, e interpretada no *shader* de forma correspondente.

Quando um *shader* de vértices é utilizado, algumas etapas do fluxo de função fixa de processamento de vértices são desativadas, e as alterações dos estados de renderização que afetam essas partes só podem ser realizadas via programação de *shaders*. São elas: 1) Transformação de coordenadas relativas ao espaço do mundo para o espaço de recorte; 2) Normalização de vetores; 3) Cálculo de iluminação; 4) Geração de coordenadas de texturas; 5) Uso de planos de recorte definidos pela aplicação (exceto em alguns ambientes de execução). Todas as demais partes do fluxo de função fixa não são substituídas. Em particular, continuam sendo válidas as seguintes etapas: 1) Montagem de primitivas; 2) Recorte de primitivas pelo volume de visualização; 3) Divisão pela coordenada homogênea; 4) Mapeamento de coordenadas na janela de visão; 5) *Back-face culling*.

O *shader* de vértices tem acesso a um conjunto de registradores de diferentes propósitos (registradores de entrada, temporários, constantes, de endereçamento e saída), com restrições de acesso ou leitura segundo o uso de cada registrador (figura 2.4). Em geral, cada registrador é um vetor com quatro componentes em ponto flutuante de até 32 bits, e as operações entre os registradores são realizadas nas quatro componentes simultaneamente. Esses valores podem ser interpretados como pontos em coordenadas homogêneas (x, y, z, w), valores de cor (r, g, b, α), coordenadas de textura (s, t, r, q) ou como 4 escalares independentes. Atualmente, os modelos de ambientes de execução de *shaders* de vértices também incluem registradores booleanos e de números inteiros.

O *shader* de vértices pode ler os atributos dos vértices a partir de até 16 registradores de en-

2007]. Isto não deve ser confundido com o termo *bitangente*, com sentido já bem definido: linha que tangencia uma curva ou superfície em dois pontos distintos. Nesta tese seguimos Lengyel [2003] e empregamos o termo *vetor bitangente*.

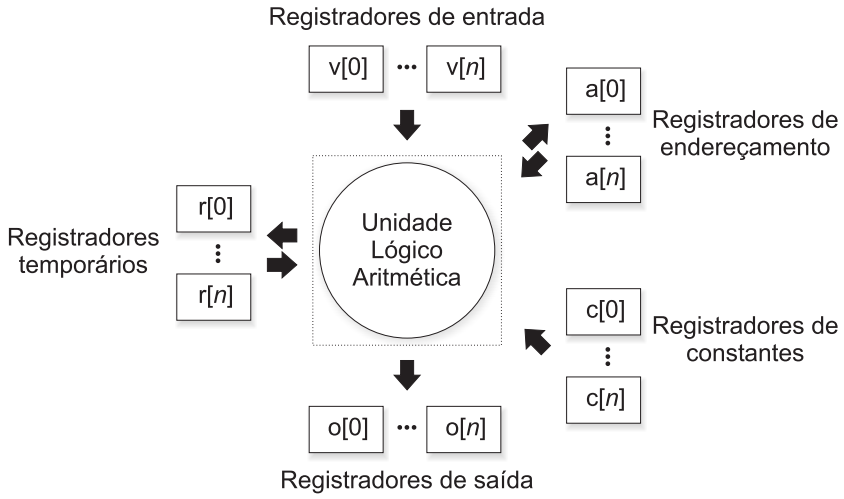


Fig. 2.4: Diagrama do ambiente geral de execução de um modificador de vértices.

trada (geralmente um registrador para cada atributo do vértice) com valores fornecidos pela etapa de montagem de vértices. O *shader* executa seus cálculos através da leitura desses registradores, usando registradores temporários para armazenar resultados intermediários do processamento. O *shader* também pode acessar registradores constantes para obter valores definidos pela aplicação e que não variam entre os vértices da geometria. Exemplos de valores usualmente armazenados em registradores constantes incluem a combinação das matrizes de transformação, o vetor de direção ou posição de uma fonte de luz, paletas de matrizes utilizadas para deformação de vértices, entre outros.

Registradores de endereçamento podem ser utilizados pelo *shader* para realizar endereçamento relativo de índices no conjunto de registradores de constantes. Esses registradores geralmente não podem ser lidos diretamente dentro do *shader*, mas usados apenas para endereçamento relativo. Ambientes de execução mais recentes, como o especificado pelo modelo de *shader* 3.0 e utilizado em nossa arquitetura de interação, utilizam registradores especiais para realizar fluxo de controle dinâmico, permitindo a implementação de procedimentos mais complexos na GPU. Uma característica especialmente útil neste modelo de *shader* é a disponibilidade, no processdor de vértices, dos estágios de amostragem de texturas exibidos na figura 2.5. Essa característica permite a realização de amostragem de texturas durante o processamento de vértices.

O *shader* de vértices armazena seus resultados em um conjunto de registradores de saída. Os registradores de saída possuem uma semântica pré-definida, tal como a posição do vértice transformado em espaço homogêneo, as coordenadas de textura e a cor do vértice. Esses resultados são transferidos para os próximos estágios do fluxo de função fixa ou utilizados como entrada do *shader* de fragmentos, e são interpolados ao longo da primitiva segundo o modelo de interpolação definido na API.

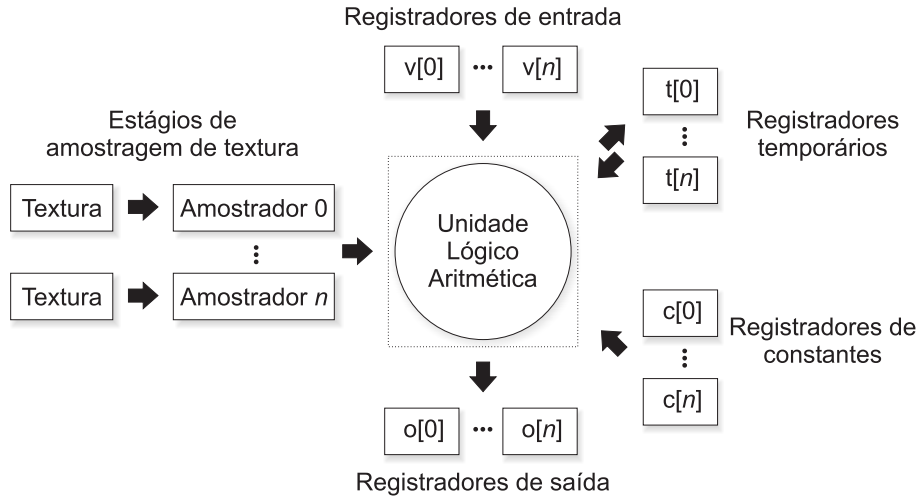


Fig. 2.5: Diagrama do ambiente geral de execução de um modificador de fragmentos.

2.4.2 Processador de fragmentos

O processador de fragmentos é responsável pela execução do *shader* de fragmentos sobre cada fragmento produzido durante a etapa de rasterização. Embora o processador de fragmentos seja algumas vezes chamado simplesmente de processador de *pixels*, um fragmento não é necessariamente um *pixel*, mas um ponto em coordenadas da tela produzido pelo rasterizador a partir da amostragem das primitivas. Desse modo, o uso de multi-amostragem pode produzir vários fragmentos para cada *pixel*, e a cor final do *pixel* poder ser a combinação das cores desses fragmentos.

Cada fragmento contém atributos associados aos vértices da saída do processador de vértices, tais como valores de cores, coordenadas de textura e atributos definidos pela aplicação. Esses atributos são interpolados a partir dos vértices segundo o modelo de interpolação escolhido. O processador de fragmentos é responsável pela determinação da cor final do fragmento e, opcionalmente, seu valor de profundidade. As tarefas do fluxo de renderização subseqüentes ao processadores de fragmentos são as mesmas do fluxo de função fixa.

Quando um processador de fragmentos é utilizado, as etapas de amostragem e combinação de texturas do fluxo de função fixa são desabilitadas. As demais funcionalidades permanecem habilitadas, tais como uso do modelo de interpolação (*flat* ou *smooth*), teste de opacidade, teste de profundidade, teste de estêncil, combinação de valores de fragmentos no *frame buffer*, *dithering* e cálculo de neblina.

O ambiente de execução de um *shader* de fragmentos é aquele mostrado na figura 2.5. Assim como no ambiente utilizado pelos *shaders* de vértices, *shaders* de fragmentos trabalham com um conjunto de registradores de entrada, saída, registradores temporários e constantes. O ambiente de execução em GPUs atuais também fornece um registrador de contagem de laços, um registrador de

predicado usado em comparações, um registrador contendo a posição do *pixel* correspondente em coordenadas da tela, e um registrador que informa a área da face que está sendo processada. Entretanto, atualmente nenhum ambiente de execução de modificadores de fragmentos utiliza registradores de endereçamento.

Os registradores de entrada contém os valores interpolados de saída do *shader* de vértices, tais como os valores de cor dos fragmentos ou coordenadas de textura, e só possuem permissão de leitura pelo modificador. Utilizando instruções especiais, o *shader* de fragmentos pode amostrar valores filtrados de textura usando estágios de amostragem de texturas.

O *shader* pode tanto usar as coordenadas interpoladas de textura passadas em um dos registradores de entrada como também as coordenadas de textura calculadas diretamente no modificador para amostrar uma textura. Usando os valores dos registradores de entrada e os valores amostrados de texturas, o *shader* produz seus resultados e armazena-os em registradores de saída. Como no *shader* de vértices, esses registradores de saída possuem uma semântica pré-definida. A saída inclui obrigatoriamente o valor final da cor do fragmento que será utilizado para formar a cor do *pixel* no *frame buffer* e, opcionalmente, o valor de profundidade que será utilizado no teste de profundidade. As atuais GPUs permitem a escrita desses valores em vários alvos de renderização ao mesmo tempo, através do uso de vários registradores de saída de cor e valor de profundidade no *shader* de fragmentos.

2.4.3 A GPU como um processador de propósito geral

Com o aumento do número de instruções aritméticas e registradores disponíveis nos processadores de vértices e fragmentos, e particularmente com o advento de instruções de fluxo de controle dinâmico e capacidade de acessar e escrever em texturas com formato em ponto flutuante, as atuais GPUs programáveis têm possibilitado a realização de processamento paralelo que extrapola aquele destinado à síntese de imagens em tempo real [Luebke et al., 2004a]. Tão importante quanto o aprimoramento do conjunto de instruções, as atuais GPUs possuem poder computacional maior do que as CPUs para processamento de fluxos de dados e, como já discutido na seção 1.1, fornecem acesso mais eficiente a dados armazenados em memória de vídeo local.

A arquitetura de uma GPU se assemelha ao modelo de um processador de fluxo (*stream processor*). Nessa arquitetura, cada elemento de um fluxo de dados é processado de forma paralela por uma mesma função, chamada *kernel*, e os resultados são armazenados em um fluxo de saída, normalmente (mas não necessariamente) com um mapeamento biunívoco entre os elementos de entrada e saída.

O processamento de cada elemento do fluxo independe do processamento dos demais elementos, possibilitando assim a criação de diversos fluxos de processamento paralelo. De fato, as atuais GPUs são compostas de centenas de fluxos simultâneos de processamento de fragmentos. Texturas podem ser consideradas como fluxo de dados de entrada e saída dispostos em matrizes 2D, nas quais os

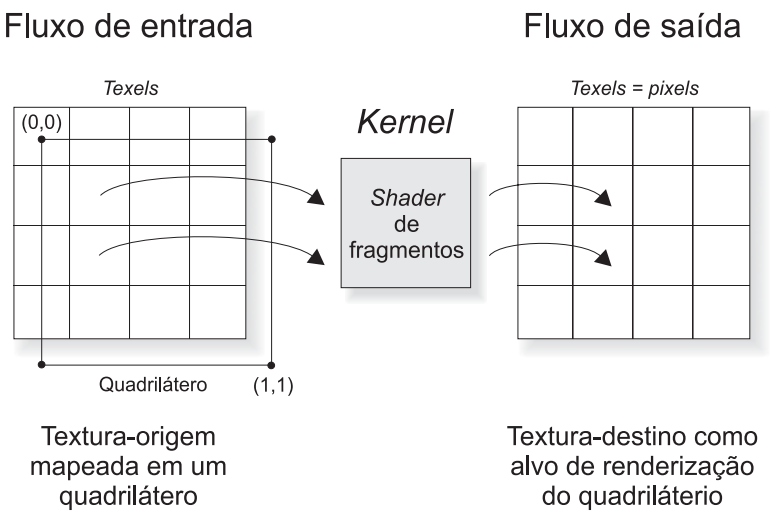


Fig. 2.6: Processamento de fluxo na GPU através da renderização de um quadrilátero que mapeia *texels* de uma textura de origem a *texels* de uma textura de destino.

elementos do fluxo são os *texels*, que de fato são vetores de até 4 valores (*e.g.*, valores $RGB\alpha$). Como os dados da textura não precisam ser interpretados como valores de cor, uma operação de amostragem de textura pode ser interpretada como uma operação de leitura de um elemento do fluxo. Da mesma forma, a operação de escrita de um valor de cor numa textura equivale à escrita de um elemento no fluxo de dados de saída. O *shader* de fragmentos, responsável pelo processamento de cada *texel*, é análogo ao *kernel* de um processador de fluxo.

O processamento de fluxos usando a GPU é normalmente realizado no processador de fragmentos, como resultado da arquitetura SIMD que permite que as GPUs implementam mais estágios de processamento paralelo de fragmentos do que de vértices. Em particular, a CPU preenche uma textura com os dados de entrada e carrega essa textura na memória de vídeo local da GPU. Essa textura representa o fluxo de entrada. A GPU é utilizada então para renderizar um retângulo em coordenadas da tela, mapeado com a textura de entrada de tal forma que cada *texel* dessa textura corresponda a apenas um *pixel* da tela. Entretanto, em vez de renderizar o resultado em um *buffer* para visualização, utiliza-se como alvo de renderização uma outra textura, considerada como o fluxo de saída. Uma vez que o retângulo já é dado em coordenadas da tela, o *shader* de vértices não realiza qualquer processamento e apenas propaga os atributos dos vértices para os próximos estágios. Esses atributos são apenas a posição transformada do vértice (em coordenadas normalizadas do dispositivo) e as coordenadas da textura interpoladas a partir das coordenadas informadas nos vértices. No *shader* de fragmentos, o valor interpolado das coordenadas de textura é utilizado para amostrar cada *texel* da textura mapeada no retângulo. O valor amostrado de cada *texel* é processado e escrito na textura de saída. Uma ilustração desta operação utilizando a renderização do quadrilátero é mostrada na figura 2.6. Os valores

	CPU	GPU
Visualização	x	x
Interação 3D	x	

Fig. 2.7: Atual lacuna na aplicação da GPU para realizar processamento vinculado à interação 3D.

$(0, 0)$ e $(1, 1)$ indicam a atribuição de coordenadas de textura nos vértices no quadrilátero.

A CPU pode transferir o conteúdo da textura de saída para a memória do sistema de modo a obter os resultados. Alternativamente, essa textura de saída pode ser utilizada como textura de entrada em outro estágio de processamento de fluxo na GPU.

2.5 Considerações finais

A flexibilidade obtida através das atuais GPUs tem sido largamente utilizada na implementação de técnicas relacionadas à visualização. Isto se deve ao poder computacional das GPUs, o qual é geralmente muitas vezes superior àquele existente nas CPUs para processamento de fluxos de dados, e à capacidade de realizar o processamento de visualização de forma paralela ao processamento de outras tarefas realizadas na CPU. Por outro lado, o potencial de suporte à implementação de tarefas de interação 3D pela GPU ainda não é devidamente explorado, seja para realizar interações que levem em consideração as modificações de geometria realizadas na GPU, seja para tornar tais interações mais eficientes em comparação com os tradicionais métodos de interação baseados em *ray picking*.

As mais difundidas arquiteturas de interface gráfica com suporte a manipulações diretas, tais como o OpenSceneGraph e OpenSG, utilizam a tradicional técnica de *ray picking* para implementar tarefas de seleção e, portanto, não tratam corretamente modelos cuja geometria tenha sido deformada na GPU. Esta não é uma limitação imposta pelo *hardware* gráfico, mas uma opção de projeto dessas arquiteturas de interface gráfica segundo o paradigma da Arquitetura Gráfica Unificada, que se mostrou bem sucedido até o advento do *hardware* gráfico programável.

Acreditamos que, se devidamente explorada, a GPU pode ser utilizada para realizar processamento relacionado a tarefas de interação 3D, preenchendo assim uma atual lacuna relacionada ao uso da GPU para interação, conforme ilustrado na figura 2.7. É sobre esta hipótese que propomos, nesta tese, uma arquitetura de suporte a tarefas de interação 3D sobre geometria processada nas GPUs.

Na arquitetura de interação proposta, a GPU é utilizada como um processador de fluxo para realizar tarefas tais como manter em memória de textura os atributos dos vértices após as modificação realizadas no processador de vértices, e para calcular elementos de geometria diferencial discreta a

partir dessa geometria modificada conforme o procedimento descrito no capítulo 4.

Capítulo 3

Atributos elementares para manipulação direta

Neste capítulo mostramos como tarefas de manipulação direta 3D usando dispositivos apontadores 2D podem ser implementadas através da leitura de atributos geométricos e não geométricos (valores definidos pela aplicação) codificados previamente como componentes de cor de cada *pixel* do modelo renderizado em *buffers* de renderização não visíveis. Esta possibilidade permite simular de forma satisfatória as tarefas de interação comumente implementadas segundo o tradicional método de *ray picking*, o qual requer a execução de testes de interseção entre um raio e cada primitiva de cada modelo. Além disso, permite levar em consideração as modificações dos atributos da geometria realizadas na GPU, caso tais atributos sejam também calculados na GPU antes de serem codificados nos *pixels*. Tal possibilidade serve de inspiração para a proposta da arquitetura de suporte a tarefas de manipulação 3D apresentada na seção 5.

Veremos que os atributos geométricos necessários para a implementação de tarefas básicas de manipulação direta 3D podem se resumir em propriedades de geometria diferencial das superfícies sob interação. Desse modo, apresentamos na seção 3.1 uma introdução a conceitos básicos de geometria diferencial de superfícies. Na seção 3.2 realizamos um estudo de casos com as tarefas básicas de manipulação 3D de modo a mostrar esse conjunto mínimo de atributos do modelo geométrico necessários para a realização de cada tarefa segundo o método tradicional de *ray picking*. Essas tarefas são os blocos básicos de construção de tarefas de manipulação direta 3D usando dispositivos apontadores 2D, e amplamente utilizadas em aplicativos de modelagem geométrica, pintura 3D e jogos. Mostramos na seção 3.3 como esse conjunto mínimo de atributos pode ser processado em um fluxo de visualização de gráficos *raster* e codificados para cada *pixel* do modelo renderizado de modo que a aplicação possa realizar a mesma tarefa básica de interação.

3.1 Geometria diferencial de superfícies

A geometria diferencial de superfícies compreende o estudo da geometria de variedades bidimensionais utilizando cálculo diferencial. Em nosso trabalho estamos especialmente interessados, para cada amostra da superfície, no cálculo de vetor normal, vetores tangentes alinhados às coordenadas de textura, a maior e menor curvatura, as direções associadas a essas curvaturas, e as derivadas dessas curvaturas.

Vamos considerar uma superfície regular descrita pela função paramétrica $S\colon R^2 \rightarrow R^3$ e um ponto $p = S(u, v)$ nessa superfície sobre a qual queremos analisar seu comportamento local em torno de p. A seguir conceituamos os elementos de geometria diferencial que utilizamos no desenvolvimento da arquitetura de suporte a tarefas de interação 3D, classificando-os segundo a ordem das derivadas de S às quais eles relacionam. Para o detalhamento desse conteúdo, sugerimos o livro-texto de Carmo [1976] e os trabalhos de Lengyel [2003] e Gravesen and Ungstrup [2002].

3.1.1 Elementos de primeira ordem

Elementos de geometria diferencial de primeira ordem são obtidos através das primeiras derivadas parciais da superfície S. Entre eles destacamos o *vetor normal*, *vetor tangente* e *vetor bitangente*. Por completude, também introduzimos o conceito de *primeira forma fundamental*.

Vetores normal, tangente e bitangente

O par de vetores de derivadas parciais $\left[\begin{array}{cc} S_u & S_v \end{array}\right] = \left[\begin{array}{cc} \frac{\partial S}{\partial u} & \frac{\partial S}{\partial v} \end{array}\right]$, juntamente com um vetor normal, define uma base local sobre o plano tangente a S em p. O vetor normal unitário N em p é definido como

$$N(u, v) = \frac{S_u \times S_v}{\|S_u \times S_v\|}. \tag{3.1}$$

Em síntese de imagens, técnicas de mapeamento de textura para simulação de detalhes 3D utilizam as coordenadas do espaço de textura (s, t) como parâmetros, havendo a necessidade de uma reparametrização $S(s, t) = S(s(u, v), t(u, v))$, de sorte que $S(s, constante)$ e $S(constante, t)$ passam a ser as curvas coordenadas de S. Assim, os vetores de base do plano tangente em qualquer ponto de S devem estar alinhados com as direções nas quais a derivada de s e t é zero. Em especial, o vetor tangente T é definido como o vetor unitário no plano tangente tal que $D_T t = 0$[1] e $D_T s > 0$. Similarmente, o vetor bitangente B é definido como o vetor unitário no plano tangente tal que $D_B s = 0$ e $D_B t > 0$. Dependendo da forma como as funções s e t são definidas, os vetores T e B não serão

[1] Derivada direcional de t na direção T.

necessariamente ortogonais e podem até mesmo não existir. Na prática, supõe-se que tais vetores sempre existem, e uma operação de ortogonalização é utilizada para forçar a obtenção de uma base ortogonal.

A relação entre T, B e o vetor normal pode ser tal que $N = T \times B$ ou $N = B \times T$ em diferentes regiões da superfície, uma vez que o produto vetorial dos vetores tangentes sempre deve apontar na mesma direção do vetor normal definido pela equação 3.1. Tal base tangente define um sistema de coordenadas que coincide com o sistema de coordenadas no qual as chamadas *texturas de normais* (*normal maps*), utilizadas em técnicas de mapeamento de detalhes 3D, são definidas. O vetor tangente corresponde ao eixo x do mapa de textura (direção crescente de s). O vetor bitangente corresponde ao eixo y (direção crescente de t), e o vetor normal aponta na direção que, intuitivamente, define qual é a "frente" da textura.

Primeira forma fundamental

A primeira forma fundamental $\mathbb{I}_s$ compreende um tensor com o qual é possível tratar questões métricas (área, comprimento e ângulo) de S sem precisar fazer referências ao espaço no qual S está imerso. Em especial, esta forma descreve a distorção da parametrização local de S em relação a vetores de base ortonormais do espaço Euclidiano. Por exemplo, um vetor tangente T a uma curva sobre S que passa por um ponto p pode ser representado em coordenadas locais por um par de coeficientes infinitesimais $U = \begin{bmatrix} \partial u & \partial v \end{bmatrix}'$ numa combinação linear dos vetores de base $\begin{bmatrix} S_u & S_v \end{bmatrix}$:

$$T = \partial u S_u + \partial v S_v = \begin{bmatrix} S_u & S_v \end{bmatrix} \begin{bmatrix} \partial u \\ \partial v \end{bmatrix}. \tag{3.2}$$

A magnitude e a direção de T variam de acordo com a magnitude e ângulo dos vetores de base. Se os vetores de base são ortonormais, a variação é a mesma do espaço $\mathbb{R}^3$. $\mathbb{I}_s$ descreve essa relação através de três coeficientes da forma quadrática obtida da quantidade $T \cdot T$:

$$
\begin{aligned}
T \cdot T &= \begin{bmatrix} S_u & S_v \end{bmatrix} \begin{bmatrix} \partial u \\ \partial v \end{bmatrix} \cdot \begin{bmatrix} S_u & S_v \end{bmatrix} \begin{bmatrix} \partial u \\ \partial v \end{bmatrix} && (3.3)\\
&= \begin{bmatrix} \partial u & \partial v \end{bmatrix} \begin{bmatrix} S_u \\ S_v \end{bmatrix} \cdot \begin{bmatrix} S_u & S_v \end{bmatrix} \begin{bmatrix} \partial u \\ \partial v \end{bmatrix} && (3.4)\\
&= \begin{bmatrix} \partial u & \partial v \end{bmatrix} \begin{bmatrix} S_u \cdot S_u & S_u \cdot S_v \\ S_v \cdot S_u & S_v \cdot S_v \end{bmatrix} \begin{bmatrix} \partial u \\ \partial v \end{bmatrix} && (3.5)\\
&= \begin{bmatrix} \partial u & \partial v \end{bmatrix} \begin{bmatrix} E & F \\ F & G \end{bmatrix} \begin{bmatrix} \partial u \\ \partial v \end{bmatrix} && (3.6)\\
&= U' \mathbb{I}_s U. && (3.7)
\end{aligned}
$$

$$
U' \mathbb{I}_s U = E\partial u^2 + 2F\partial u\partial v + G\partial v^2. \quad (3.8)
$$

É possível assumir que S é parametrizada por coordenadas curvilíneas ortonormais, de sorte que $\mathbb{I}_s$ em sua forma matricial é sempre uma matriz identidade ($E = G = 1, F = 0$). Esse pressuposto simplifica o cálculo de elementos de segunda ordem e é razoável caso as quantidades de interesse sejam apenas elementos tais como curvaturas normais, curvatura média, curvatura Gaussiana, curvaturas principais e direções principais, pois estas quantidades independem do tipo de parametrização. De fato, em malhas triangulares e nuvens de pontos, não há informação sobre curvas coordenadas e qualquer parametrização não degenerada pode ser assumida.

3.1.2 Elementos de segunda ordem

Segundo as equações de Weingarten [Carmo, 1976], a derivada direcional $D_T N$ de N pode ser definida como a combinação linear de $\begin{bmatrix} S_u & S_v \end{bmatrix}$ com os coeficientes infinitesimais $\begin{bmatrix} N_u & N_v \end{bmatrix} = \begin{bmatrix} \frac{\partial N}{\partial u} & \frac{\partial N}{\partial v} \end{bmatrix}$.

A *curvatura normal* define o quanto N é rodado a partir de um deslocamento tangencial infinitesimal sobre uma seção normal de S, seção esta obtida por um plano paralelo a N, contendo p, que intersecta S na direção T. A interseção entre esse plano e S forma uma curva na direção T cuja curvatura no ponto p equivale ao valor recíproco do raio ρ do círculo osculador que passa por este ponto: $\kappa_T(p) = \frac{1}{\rho}$.

A segunda forma fundamental $\mathbb{II}_s$ está relacionada à informação sobre o quanto um vetor N normal à S no ponto p varia numa direção tangencial à curva que passa por esse ponto, ou seja, à derivada direcional $D_T N$ de N numa direção T

$$
\begin{aligned}
-T \cdot D_T N &= \begin{bmatrix} \partial u & \partial v \end{bmatrix} \begin{bmatrix} -S_u \\ -S_v \end{bmatrix} \cdot \begin{bmatrix} N_u & N_v \end{bmatrix} \begin{bmatrix} \partial u \\ \partial v \end{bmatrix} && (3.9)\\
&= \begin{bmatrix} \partial u & \partial v \end{bmatrix} \begin{bmatrix} -S_u \cdot N_u & -S_u \cdot N_v \\ -S_v \cdot N_u & -S_v \cdot N_v \end{bmatrix} \begin{bmatrix} \partial u \\ \partial v \end{bmatrix} && (3.10)\\
&= \begin{bmatrix} \partial u & \partial v \end{bmatrix} \begin{bmatrix} e & f \\ f & g \end{bmatrix} \begin{bmatrix} \partial u \\ \partial v \end{bmatrix} && (3.11)\\
&= U' \mathbb{II}_s U. && (3.12)
\end{aligned}
$$

$$
U' \mathbb{II}_s U = e\partial u^2 + 2f\partial u \partial v + g\partial v^2. \quad (3.13)
$$

A simetria de $\mathbb{II}_s$ é obtida através da observação que $N \cdot S_u = N \cdot S_v = 0$. $\mathbb{II}_s$ também pode ser expressa por produtos escalares entre N e derivadas de segunda ordem

$$
\mathbb{II}_s = \begin{bmatrix} e & f \\ f & g \end{bmatrix} = \begin{bmatrix} S_{uu} \cdot N & S_{uv} \cdot N \\ S_{vu} \cdot N & S_{uv} \cdot N \end{bmatrix}, \quad (3.14)
$$

observando que

$$
\begin{aligned}
-S_u \cdot N_u &= S_{uu} \cdot N. && (3.15)\\
-S_u \cdot N_v &= S_{uv} \cdot N. && (3.16)\\
-S_v \cdot N_u &= S_{vu} \cdot N. && (3.17)\\
-S_v \cdot N_v &= S_{uv} \cdot N. && (3.18)
\end{aligned}
$$

A matriz $W = \mathbb{I}_S^{-1} \mathbb{II}_s$, onde $\mathbb{I}_S^{-1}$ é a inversa de $\mathbb{I}_s$, é conhecida como operador de forma, ou operador de Weingarten. Considerando vetores de base ortonormais, $W = \mathbb{II}_s$. Essa matriz pode ser transformada em uma matriz diagonal através de uma rotação do sistema de coordenadas locais. Os elementos resultantes da diagonal (*i.e.*, os autovalores) correspondem às curvaturas normais mínima e máxima $\kappa_{min}, \kappa_{max}$. Curvaturas em quaisquer outras direções são combinações convexas destas duas curvaturas. Os autovetores correspondentes são denominadas as direções principais $S_{\kappa_{min}}, S_{\kappa_{max}}$ no plano tangente e indicam a direção das curvas com curvatura mínima e máxima que passam por p. A equação dos autovetores é degenerada nos chamados *pontos umbilicais*, onde $\kappa_{min} = \kappa_{max}$. Nesses pontos, todas as direções do plano tangente ao redor de p são principais, e indicam que a superfície é

localmente esférica.

A multiplicação $\mathbb{II}_s(U,U) = U'\mathbb{II}_s U$ (considerando que $W = \mathbb{II}_s$) de qualquer vetor unitário U em coordenadas locais do plano tangente, produz um escalar. Tal valor é a curvatura normal $\kappa_U(p)$ na direção U, no ponto p onde $\mathbb{II}_s$ foi calculado. Por sua vez, a multiplicação $\mathbb{II}_s(U) = \mathbb{II}_s U$ produz um vetor no plano tangente que equivale à derivada direcional de N na direção U ($D_U N$).

A partir de κ_{min} e κ_{max} é possível definir as curvaturas *Gaussiana* e *média*, utilizadas para avaliar a superfície de forma qualitativa. O valor $K = \kappa_{min}\kappa_{max}$, conhecido como *curvatura Gaussiana*, indica se a superfície é localmente elíptica ($K > 0$), hiperbólica ($K < 0$) ou parabólica/plana ($K = 0$). A *curvatura média* é definida como $H = \frac{1}{2}(\kappa_{min} + \kappa_{max})$, e está relacionada à área delimitada por um contorno fechado da superfície. Dada uma curva fechada, a superfície de menor área delimitada por ela tem curvatura média igual a zero em todos os pontos. Tal superfície é conhecida como superfície mínima.

3.1.3 Elementos de terceira ordem

Gravesen e Ungstrup [Gravesen and Ungstrup, 2002] definem um tensor $\mathbb{C}_s$ relacionado à taxa de mudança da curvatura de S ao redor de p. Através deste tensor é possível obter as derivadas das curvaturas normais. $\mathbb{C}_s$ é um tensor simétrico trilinear $2\times2\times2$ (uma matriz 3D) de posto 3, composto de quatro elementos únicos baseados em derivadas parciais de terceira ordem. Organizado como um vetor de matrizes, $\mathbb{C}_s$ tem a seguinte forma:

$$\mathbb{C}_s = \begin{bmatrix} D_u\mathbb{II}_s & D_v\mathbb{II}_s \end{bmatrix} = \left[\begin{bmatrix} a & b \\ b & c \end{bmatrix}\begin{bmatrix} b & c \\ c & d \end{bmatrix}\right], \tag{3.19}$$

onde

$$\begin{aligned} a &= S_{uuu}\cdot N + 3S_{uu}\cdot N_u, && (3.20)\\ b &= S_{uuv}\cdot N + S_{uu}\cdot N_v + 2S_{uv}\cdot N_u && (3.21)\\ &= f_u + S_{uu}\cdot N_v + S_{uv}\cdot N_u, && (3.22)\\ c &= S_{uvv}\cdot N + S_{vv}\cdot N_u + 2S_{uv}\cdot N_v && (3.23)\\ &= f_v + S_{vv}\cdot N_u + S_{uv}\cdot N_v, && (3.24)\\ d &= S_{vvv}\cdot N + 3S_{vv}\cdot N_v, && (3.25) \end{aligned}$$

e f é o segundo componente de $\mathbb{II}$. Se as curvaturas e direções principais forem conhecidas, os valores

de $\mathbb{C}_s$ podem ser obtidos facilmente através das derivadas direcionais de κ_{min} e κ_{max} nessas direções principais:

$$a = D_{S_{\kappa_{min}}}\kappa_{min}, \tag{3.26}$$
$$b = D_{S_{\kappa_{max}}}\kappa_{min}, \tag{3.27}$$
$$c = D_{S_{\kappa_{min}}}\kappa_{max}, \tag{3.28}$$
$$d = D_{S_{\kappa_{max}}}\kappa_{max}. \tag{3.29}$$

A multiplicação de $\mathbb{C}_s$ por um vetor unitário $U = \begin{bmatrix} \partial u & \partial v \end{bmatrix}$ no plano tangente produz um tensor simétrico 2×2 que corresponde à derivada direcional de $\mathbb{II}_s$ na direção U:

$$\mathbb{C}_S(U) = D_U\mathbb{II}_s. \tag{3.30}$$

Ao multiplicar $\mathbb{C}_s$ por U duas vezes, obtém-se o vetor gradiente da curvatura normal na direção U (pela simetria, a ordem da multiplicação não altera o resultado):

$$\mathbb{C}_S(U,U) = \nabla\kappa_U = g_u S_{\kappa_{min}} + g_v S_{\kappa_{max}}, \tag{3.31}$$

onde

$$g_u = a\partial u^2 + 2b\partial u\partial v + c\partial v^2, \tag{3.32}$$
$$g_v = b\partial u^2 + 2c\partial u\partial v + d\partial v^2. \tag{3.33}$$

Por fim, a multiplicação de $\mathbb{C}_s$ por U três vezes produz um escalar que corresponde à derivada da curvatura na direção U:

$$\mathbb{C}_s(U,U,U) = D_U\kappa_U = a\partial u^3 + 3b\partial u^2\partial v + 3c\partial u\partial v^2 + d\partial v^3. \tag{3.34}$$

3.2 Estudo de casos

Em manipulação direta, tarefas compostas de interação são construídas a partir de uma seqüência de tarefas básicas de interação. Em aplicações gráficas 2D, as tarefas básicas de posicionamento e seleção são geralmente de fácil implementação. Para o posicionamento, o sistema de janelas provê automaticamente as coordenadas do posicionamento do cursor do dispositivo apontador no referencial da tela. A aplicação precisa apenas mapear tais coordenadas para o sistema de coordenadas convencionado para o referencial do aplicativo. Para a tarefa de seleção, a aplicação realiza um teste de inclusão de um ponto 2D em uma região. Em geral, esses gráficos 2D não sofrem modificações de geometria nas GPUs, e todo o processamento pode ser realizado na CPU. Para interações 3D realizadas através de dispositivos apontadores 2D, essas tarefas básicas são as tarefas de posicionamento de um cursor 2D ou cursor 3D controlado pelo dispositivo apontador, e seleção de elementos 3D da cena apontados por um cursor 2D.

Nesta seção vamos analisar, através de um estudo de casos, o conjunto de dados que são, de fato, essenciais para interações 3D. Em particular, descrevemos os conjuntos de dados que devem ser obtidos para diferentes tarefas de manipulação 3D, com o objetivo de produzir interações eficientes e consistentes com modelos cujos atributos são modificados em processadores de vértices ou de fragmentos das atuais GPUs.

3.2.1 Seleção 3D

A tarefa de seleção usando um dispositivo apontador 2D é geralmente realizada através da técnica de *ray picking*. Ela consiste em gerar um raio, em coordenadas do espaço do mundo, que parte da posição da câmera e passa pela posição do ponto indicado pelo cursor. Um teste de interseção é realizado entre esse raio e todos os modelos armazenados no banco de dados da cena. O modelo selecionado é o modelo cujo ponto de interseção com o raio se encontra mais próximo do plano de projeção [Foley et al., 1990]. Neste teste de interseção, são retornados o identificador do modelo intersectado e, opcionalmente, dados geométricos associados ao ponto de interseção, como a posição 3D e vetor normal à superfície naquele ponto [Mic, 2005]. Com esses dados é possível realizar tarefas de seleção, mas também posicionamento de um cursor 3D naquele ponto. Uma vez que o teste de interseção é realizado na CPU, deformações de geometria ocorridas durante o fluxo de renderização não são levadas em consideração (figura 3.1).

Uma abordagem muito comum para adaptar *ray picking* à geometria deformada na GPU consiste em implementar uma cópia das funções de deformação na CPU e utilizá-las sempre que uma interação com o modelo for necessária. Essa estratégia tem sido utilizada extensivamente por desenvolvedores de jogos para implementar a tarefa de seleção de malhas animadas por *mesh skinning* [Use,

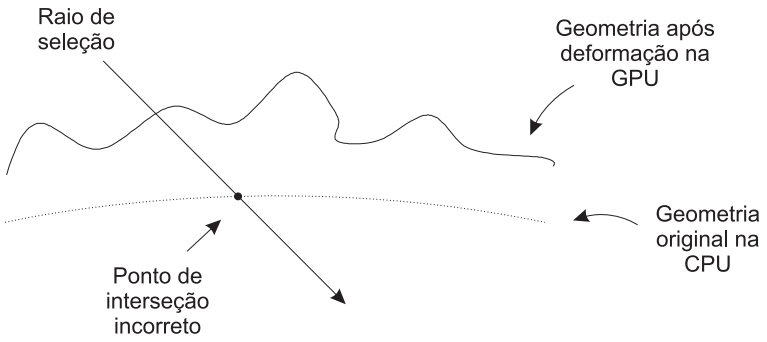


Fig. 3.1: Estimativa incorreta do ponto de interseção entre a geometria deformada na GPU e o raio de seleção calculado com *ray picking* avaliado na CPU.

2002, 2004]. Neste tipo de animação, um modelo estático é submetido à GPU, juntamente com uma hierarquia de matrizes de transformação e atributos de vértices contendo valores de ponderação da influência de cada matriz. Em tempo de execução, apenas as matrizes são modificadas e enviadas novamente à GPU. O processador de vértices aplica tais transformações a cada vértice de acordo com os valores de ponderação associados, sem precisar recorrer à CPU. O resultado é uma geometria deformada, enviada diretamente para renderização. Para interação, este procedimento é repetido na CPU de modo a obter o mesmo resultado de deformação, porém sobre o modelo armazenado na memória do sistema. Tal abordagem prejudica o desempenho da aplicação em razão da necessidade de processar a geometria duas vezes durante a interação: uma para visualização e outra para interação. Outra abordagem, igualmente ineficiente, consiste em deixar de utilizar a GPU para deformação de geometria nos momentos em que o processamento para interação é exigido. Nesses casos, a aplicação deve enviar a geometria já modificada à GPU, o que gera um gargalo no barramento. Mais do que isso, tanto a abordagem de realizar o cálculo duas vezes (na CPU e na GPU), como a de ignorar o processamento na GPU durante a interação, são inviáveis quando se deseja trabalhar com deformação de geometria em seu nível de fragmentos, pois este caso implicaria a implementação, na CPU, de todo o estágio de processamento de fragmentos.

A API OpenGL fornece uma abordagem alternativa para a tarefa de seleção e posicionamento através de um modo especial de renderização chamado *modo de seleção* [Shreiner et al., 2005]. No modo de seleção, fragmentos de *pixels* não são gerados a partir da rasterização. Em vez disso, se a geometria produz uma interseção com um "volume de recorte" definido pelo volume de visualização e por planos de recorte definidos pela aplicação, um evento de seleção (*selection hit*) é gerado e retornado para a aplicação. Uma ilustração desse procedimento é mostrada na figura 3.2. Em geral, o volume de recorte utilizado no modo de seleção é configurado de forma semelhante ao volume de visualização, porém envolvendo apenas o *pixel* apontado pelo cursor. Cada *selection hit* contém

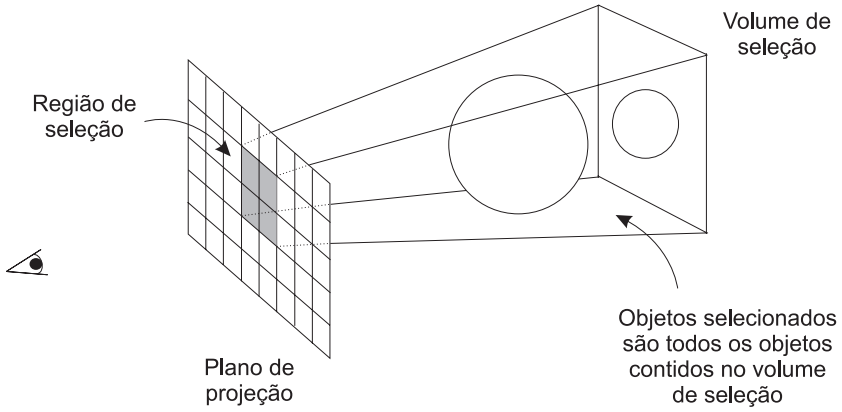


Fig. 3.2: Seleção usando o *modo de seleção* do OpenGL.

dados associados ao evento de interseção entre a geometria e o volume de recorte, tais como um identificador numérico definido pela aplicação e utilizado para identificar a geometria intersectada, e valores mínimos e máximos de profundidade da geometria dentro desse volume de recorte. Tais dados são suficientes para que a aplicação determine o objeto selecionado e a posição 3D correspondente ao ponto de seleção. Esse procedimento é realizado sem requerer processamento em um banco de dados da cena na CPU. Para determinar tais dados, o OpenGL utiliza as próprias primitivas enviadas pela aplicação para o fluxo de renderização.

Explorando as características do OpenGL, mostramos em [Wu et al., 2003] a possibilidade de realizar seleção 3D com o uso do modo de seleção (para selecionar objetos convexos) e o *buffer* estêncil (para selecionar objetos não convexos), tornando desnecessário o uso do algoritmo de *ray picking* com a existência do modelo geométrico na CPU. Nesta ocasião, exploramos a idéia utilizar um conjunto mínimo e suficiente de propriedades geométricas (posição 3D e vetor normal) do modelo para realizar a tarefa de seleção e posicionamento restrito.

Uma vez que os *selection hits* são calculados após o processamento de vértices, o modo de seleção leva em consideração a geometria modificada no estágio de processamento de vértices da GPU. Entretanto, a modificação de fragmentos no processador de fragmentos não é considerada, já que a seleção é avaliada antes da rasterização. Além disso, não é possível obter dados geométricos adicionais sobre os pontos de interseção, como o vetor normal à superfície nesses pontos. Em [Wu et al., 2003], o vetor normal ainda é calculado pela aplicação.

Atualmente, as GPUs contam com uma funcionalidade de consulta de visibilidade de fragmentos, conhecida como *hardware occlusion queries* [Microsoft, 2006]. Ao habilitar essa consulta, a GPU calcula, para todas as primitivas enviadas ao fluxo de visualização, a quantidade de fragmentos que passam pelo teste de profundidade e estão de fato visíveis na tela. Esse processamento é realizado de

forma assíncrona com o processamento na CPU, o que possivelmente aumenta o desempenho geral da aplicação, uma vez que esta pode realizar outras tarefas enquanto a consulta de visibilidade está sendo realizada. Esta funcionalidade pode ser utilizada para implementar um algoritmo de seleção eficiente que leva em consideração tanto a deformação de geometria com relação aos vértices como com relação aos fragmentos produzidos no rasterizador. Inicialmente, as dimensões do volume de visualização são reduzidas de modo a envolver apenas o *pixel* apontado pelo cursor. Todos os objetos são renderizados em um *buffer* de renderização não visível, com escrita habilitada no *buffer* de profundidade. Os objetos são então renderizados mais uma vez, desta vez sem escrita ao *buffer* de profundidade, mas com o teste de profundidade habilitado de modo a descartar fragmentos que estão sendo encobertos por objetos já renderizados. Para cada objeto renderizado, habilita-se a consulta de visibilidade de modo a verificar se algum fragmento foi renderizado. Os objetos selecionados serão aqueles que produziram fragmentos visíveis segundo o teste de visibilidade. Embora seja eficiente, essa abordagem não é capaz de retornar informações adicionais sobre os pontos visíveis da superfície, como posição 3D e vetor normal. Dessa forma, não pode ser utilizada para implementação de tarefas de posicionamento com restrição e prover realimentação visual apropriada (seção 3.2.2).

Apesar das diversas formas para realizar uma seleção, o método de *ray picking* é ainda popular em aplicações de modelagem geométrica, animação, pintura 3D e jogos. Esta popularidade se deve ao fato dele também permitir retornar todos os objetos intersectados pelo raio de seleção, incluindo os objetos não visíveis ao observador, na forma de um conjunto de valores que identificam tais objetos. Dentre os principais aplicativos de modelagem geométrica que adotam a estratégia de *ray picking* podemos citar Autodesk AutoCAD [Autodesk, 2007b], Autodesk 3ds Max [Autodesk, 2007a] e Autodesk Maya [Autodesk, 2007c]. Em relação aos principais aplicativos de pintura 3D temos o Right Hemisphere Deep Creator/Deep Paint 3D [Hemisphere, 2007], Pixologic ZBrush [Pixologic, 2007] e Interactive Effects Amazon Paint [Effects, 2007]. Seu uso também é extensivo em jogos, uma vez que as principais bibliotecas de suporte ao desenvolvimento de jogos, chamadas de *game engines*, disponibilizam funcionalidades de interação baseadas em testes de interseção. Nesta categoria estão incluídas bibliotecas tais como o Torque Game Engine [GarageGames, 2007], 3D GameStudio [Datasystems, 2007], Irrlicht Engine [Gebhardt, 2007], além das bibliotecas de grafo de cena tais como o OpenSceneGraph [OpenSceneGraph, 2007] e OpenSG [OpenSG, 2007].

Do nosso estudo, verificamos que os únicos atributos necessários para realizar uma tarefa de seleção usando um dispositivo apontador 2D são os valores de identificação dos modelos renderizados. Esses valores devem ser disponibilizados nos *pixels* associados à posição corrente do cursor 2D.

3.2.2 Posicionamento

O uso de manipulação direta em cenas 3D se difundiu na década de 1980 [Hand, 1997]. Além da seleção dos objetos de interesse numa cena, poder posicionar de forma precisa o cursor sobre um ponto específico em uma cena 3D é também fundamental. O posicionamento de um cursor pode ser livre ou restrito. No posicionamento livre o cursor controlado navega livremente pela cena sem levar em consideração os modelos geométricos, *i.e.*, o conteúdo da cena não interfere na movimentação do cursor. Em geral, essa tarefa se resume na realização de um mapeamento entre a movimentação do dispositivo apontador e a movimentação do cursor no espaço 3D de interesse. Entre as técnicas encontradas na literatura podemos mencionar a proposta de Nielson and Dan R. Olsen [1987] e Bier [1987].

Na tarefa de posicionamento restrito, o cursor controlado pelo dispositivo apontador tem sua movimentação restrita a elementos da cena tais como superfícies, linhas e pontos, de modo a aumentar o grau de precisão de posicionamento do cursor pelo usuário. Para tanto, funções adicionais são incluídas para ajustes finos da posição do cursor de sorte que ele fique sempre nos locais pré-programados. Por exemplo, Hudson [1990] se baseou em critérios semânticos, não geométricos, para restringir o cursor a objetos de um ambiente de programação visual 2D. Gleicher [1995] introduziu o conceito de restrição de cursor em imagens (*image snapping*) empregando métodos de detecção de bordas para obter pontos de restrição que facilitassem tarefas de segmentação de imagens e traçado de curvas a partir de imagens digitalizadas.

No entanto, a técnica mais tradicional para posicionamento restrito também é baseada no método de *ray picking*. Ela é utilizada, por exemplo, em aplicativos de modelagem geométrica tais como o Autodesk AutoCAD [Autodesk, 2007b], Autodesk 3ds Max [Autodesk, 2007a] e Autodesk Maya [Autodesk, 2007c]. Como vimos na seção 2.2, o algoritmo de *ray picking* se reduz à determinação de atributos de geometria associados às interseções entre o raio de seleção e os modelos geométricos em cada interação. Utilizando o método de *ray picking*, o posicionamento restrito a uma superfície consiste em calcular as posições 3D de todas as interseções entre o raio de seleção e os modelos geométricos. O cursor 3D é então deslocado para a interseção mais próxima do plano de projeção. Em uma tarefa de posicionamento restrito segundo a influência de um campo de atração, o algoritmo de *ray picking* pode ser modificado de modo a calcular o ponto visível da superfície que possua a menor distância entre o raio de seleção e cada modelo. Se essa distância for menor do que a distância determinada pelo campo de atração, o cursor pode ser deslocado para esse ponto, simulando assim o efeito de atração desejado.

Bier [1987], em sua técnica de *snap-dragging*, introduziu a idéia de utilizar um *widget* 3D - o *skitter* - de modo a auxiliar o usuário na realização de tarefas de construção através de uma metáfora de régua e compasso. O *skitter*, também chamado de *cursor tríade* [Nielson and Dan R. Olsen,

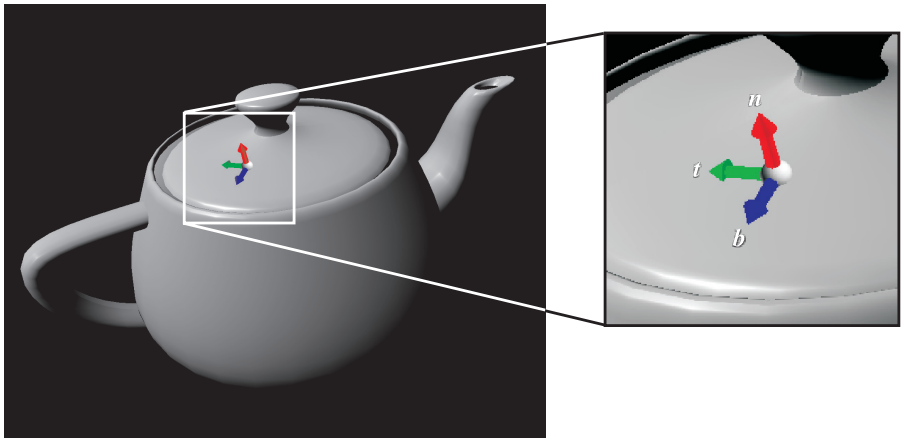


Fig. 3.3: Cursor tríade sobre uma superfície. O eixo vermelho indica a direção do vetor normal ($\mathbf{n}$). Os dois outros eixos são alinhados de acordo com duas direções tangentes ($\mathbf{t}, \mathbf{b}$) na superfície.

1987], é um cursor 3D controlado por um dispositivo apontador 2D. Quando habilitado por funções de atração, o posicionamento desse cursor é restrito a pontos, curvas ou superfícies da cena. O cursor tríade pode ser uma representação gráfica de uma base tangente, visualizado como três segmentos de linha (ou cilindros) mutuamente ortogonais que se encontram em um ponto comum. Dois segmentos são dispostos no plano tangente da superfície de interesse e o terceiro é alinhado segundo a direção do vetor normal à superfície (figura 3.3). O cursor tríade fornece ao usuário uma melhor percepção visual da posição 3D e orientação da primitiva geométrica sobre a qual o ponto foi restrito. Os dados necessários para calcular a posição e a orientação do cursor tríade são obtidos através de consultas ao banco de dados da cena.

Note que, fixadas a posição e a direção do eixo do cursor tríade correspondente ao vetor normal, há ainda um grau de liberdade para orientar o cursor sobre a superfície. Dependendo das aplicações, restrições adicionais podem ser impostas. Por exemplo, os vetores no plano tangente podem ser alinhados de acordo com a parametrização do modelo representado por uma função paramétrica, ou de acordo com a parametrização das coordenadas de textura. A disponibilidade de uma base tangente alinhada com as coordenadas de textura é comumente necessária em técnicas de mapeamento de detalhes 3D – técnicas de texturização executadas no processador de fragmentos para simular detalhes de mesoestrutura na superfície de um modelo (*e.g.*, *normal mapping* [Blinn, 1978], *parallax mapping* [Welsh, 2004] e *relief mapping* [Policarpo et al., 2005]). Em tarefas de manipulação direta para pintura 3D, o alinhamento do cursor 3D de acordo com o mapeamento das coordenadas de textura fornece ao usuário uma melhor realimentação visual da parametrização dessas coordenadas sobre a superfície.

O alinhamento dos vetores no plano tangente também pode ser realizado de acordo com as pro-

priedades de geometria diferencial da superfície. Propriedades de geometria diferencial, tais como as curvaturas e suas derivadas, são comumente utilizadas em aplicações de suavização anisotrópica de modelos digitalizados [Westgaard and Nowacki, 2001, Lange and Polthier, 2005], extração de características de forma [Ohtake et al., 2004, Weidenbacher et al., 2005] e mapeamento de detalhes 3D com renderização correta das silhuetas dos detalhes [Wang et al., 2003]. Em interação, as direções do plano tangente associadas às curvaturas mínima e máxima podem ser utilizadas em pintura 3D para guiar os traços do usuário de modo a transmitir melhor a percepção da forma do modelo, como ocorre nas técnicas de *hatching* com coerência no espaço do objeto [Elber, 1999, Praun et al., 2001]. Regiões com curvatura zero na direção de visão, mas ao mesmo tempo intersectadas com as regiões nas quais a derivada da curvatura nessa direção é positiva, formam os chamados *contornos sugestivos*, utilizados em renderização não fotorealista [DeCarlo et al., 2003, 2004].

Na aplicação de cálculo de contornos sugestivos, o tensor $\mathbb{C}_s$ pode ser utilizado para obter a derivada direcional da chamada *curvatura radial* κ_r. A curvatura radial é definida como a curvatura normal na direção do vetor formado entre p e o observador, considerando a projeção deste vetor no plano tangente. De acordo com DeCarlo et al. [2004], a derivada direcional da curvatura radial é dada por

$$D_w \kappa_r = \mathbb{C}_s(w, w, w) + 2K \cot \theta, \tag{3.35}$$

onde w é a projeção do vetor de visão sobre o plano tangente, K é a curvatura Gaussiana e θ é o ângulo entre o vetor normal e o vetor de visão. Os contornos sugestivos ocorrem nos pontos onde simultaneamente $D_w \kappa_r > 0$ e $\kappa_r = 0$. Este último valor pode ser obtido a partir da segunda forma fundamental: $\kappa_r = \mathbb{II}_s(w, w)$.

Em uma variação do posicionamento restrito a superfícies, o cursor 3D pode ser restrito a vértices e arestas do modelo representado por uma malha triangular. Em tais casos, é comum a utilização de funções de simulação de campo de atração para produzir um efeito de atração do cursor 3D à medida que este se aproxima do elemento de interesse. Esta funcionalidade, inaugurada pelo sistema Sketchpad [Sutherland, 1963], é encontrada em praticamente todos os aplicativos de modelagem geométrica e pintura 3D atuais. No método de *ray picking*, a simulação do campo de atração em relação a uma entidade de dimensão menor é realizada de forma semelhante à simulação de gravidade em relação à superfície: calcula-se, para cada elemento (*e.g.*, vértice, aresta), o ponto que possui a menor distância com relação ao raio de seleção. Se essa distância for menor do que a distância determinada pelo campo de gravidade, o cursor 3D é deslocado até este ponto, mas somente se este ponto for visível ao observador. Este processamento pode ser bastante custoso para uma cena composta de milhões desses elementos. Em tais casos, torna-se essencial o uso de algum tipo de subdivisão espacial hierárquica

para descartar rapidamente partes da cena que não são intersectadas pelo campo de gravidade do raio de seleção.

O posicionamento pode ser feito com relação a outras características do modelo geométrico. Por exemplo, Yoo and Ha [2004] utilizam magnitudes de curvaturas para calcular uma função de simulação de campo de gravidade capaz de simular a atração do cursor para vales ou topos da superfície. Estes lugares geométricos da superfície podem ser determinados pelas derivadas de sua equação paramétrica, calculadas analiticamente ou estimadas a partir da malha triangular do modelo. Para o posicionamento restrito a vales ou topos da superfície usando a estratégia de codificação de atributos em *pixels* da imagem, o modelo é renderizado com primitivas preenchidas e, para cada *pixel* do modelo, armazenam-se informações referentes às estimativas das curvaturas da superfície discretizada, além do valor de profundidade. Para cada *pixel* contido na região em torno do *pixel* que contém o cursor, a aplicação utiliza tais curvaturas para verificar quais os *pixels* contém os mínimos ou máximos locais da superfície naquela região (*e.g.*, segundo o método empregado por Yoo and Ha [2004]). Assim, o cursor 2D é deslocado para o *pixel* com o mínimo ou máximo local mais próximo da posição atual do cursor e o cursor 3D é deslocado para a posição 3D correspondente sobre a superfície.

A tarefa de posicionamento restrito também tem uso em aplicações que envolvem renderizações baseadas em imagens. Nessas aplicações, pode ser desejável restringir o movimento do cursor a características da imagem visualizada, aumentando assim o grau de precisão do movimento do cursor. Para que o usuário possa realçar determinadas partes do modelo, essas características podem ser, por exemplo, os contornos do modelo, ou detalhes obtidos do mapeamento de texturas sobre o modelo renderizado. Para este último caso, o procedimento de restrição pode ser análogo ao procedimento proposto por Hanrahan and Haeberli [1990] para pinturas 3D. Nestes casos, o método de *ray picking* não funcionaria, uma vez que este trabalha apenas com a representação geométrica dos modelos no banco de dados da cena.

Sintetizando, os dados essenciais para posicionamento são extraídos dos atributos geométricos do modelo sobre o qual o posicionamento é realizado. É importante ressaltar aqui que, em todos os casos analisados, não são empregadas modificações da geometria ou da imagem em *hardware* gráfico. Todas as alterações da geometria ou imagem são realizadas na CPU.

3.3 Uma estratégia alternativa à técnica *ray picking*

Utilizando a estratégia de *ray picking*, vimos que tanto o problema de seleção quanto o de posicionamento se reduzem algebricamente à interseção entre um raio e modelos geométricos. Portanto, o resultado depende da eficiência e do domínio de cobertura dos algoritmos de interseção. Além disso, ela não se aplica a cenas geradas com técnicas de renderização baseadas em imagens. Desse

modo, discutimos nesta seção um método alternativo para controle preciso do cursor.

A determinação algébrica das interseções pode ser contornada com a estratégia relacionada à codificação de atributos em *pixels* da imagem. Por exemplo, a seleção pode utilizar a mesma idéia da tradicional técnica de *buffer* de itens (*item buffer*) [Weghorst et al., 1984], *i.e.*, cada modelo geométrico é renderizado com uma cor distinta em um *buffer* de renderização não visível, e a cor obtida do *pixel* apontado pelo cursor identifica o modelo selecionado. Em resumo, o único atributo necessário para esta tarefa consiste em um valor identificador que indica qual modelo foi renderizado em cada *pixel*.

O procedimento de seleção de todos os modelos intersectados pelo raio de seleção também pode ser realizado através da estratégia de codificação de atributos em *pixels* da imagem, sem recorrer a cálculos de interseção. Para obter o identificador de todos os modelos que teriam interseções com o raio de seleção determinado pelo cursor 2D, sugerimos o seguinte procedimento iterativo: (1) seleciona-se o modelo visível segundo o método de *buffer* de itens; (2) renderiza-se a cena novamente no *buffer* de renderização não visível, com exceção do(s) modelo(s) já selecionado(s); (3) repete-se o procedimento a partir do primeiro passo até que nenhum identificador seja encontrado no *pixel* de interesse. Essa seqüência de passos resulta em um conjunto de identificadores dos modelos que teriam intersectado o raio de seleção segundo o método de *ray picking*, em uma ordem de frente para trás a partir do plano de projeção.

O procedimento de seleção de todos os modelos intersectados pelo raio de seleção pode ser estendido para o procedimento de selecionar todas as faces intersectadas pelo raio de seleção. Em cada iteração das etapas descritas anteriormente, são excluídas as faces já selecionadas. Para isso, cada face precisa ter seu identificador próprio (*e.g.*, uma cor própria). Nesse caso, em um modelo descrito por uma malha triangular, deve-se utilizar geometria não indexada, pois vértices comuns a vários triângulos terão identificadores de face diferentes para cada face.

Na estratégia de codificação de atributos em *pixels* da imagem, a restrição a superfícies pode ser realizada de forma semelhante à seleção, *i.e.*, obtendo informações codificadas no *pixel* apontado pelo cursor 2D. Para posicionar e orientar o cursor 3D restrito sobre a superfície, são necessários os atributos de posição 3D, vetor normal e vetores tangentes à superfície no ponto de restrição. O cálculo dessas propriedades pode ser realizado de forma analítica (quando a equação paramétrica da superfície é conhecida) ou estimado a partir do modelo discretizado. Caso o modelo seja transformado apenas por transformações de corpo rígido, este processamento pode ser realizado em uma etapa de pré-processamento. Caso contrário, deve ser feito em tempo de execução, o que pode ser custoso.

Se considerarmos superfícies discretizadas cuja renderização é realizada com as atuais GPUs, sugerimos utilizar o próprio fluxo de visualização para calcular de forma eficiente os atributos necessários à restrição a superfícies. No processador de vértices calcula-se o valor de profundidade do

vértice atual, armazenando-o como um valor de componente de cor do vértice resultante. Durante a rasterização, esse valor é interpolado linearmente para cada fragmento da primitiva, e armazenado para cada *pixel* da primitiva rasterizada. A aplicação pode ler esse valor de profundidade e, juntamente com os dados de configuração da janela de visualização e matrizes de transformação, determinar a posição 3D correspondente. O cursor 3D é então deslocado para essa posição. Um procedimento semelhante é aplicado para obter as informações do vetor normal e vetores tangentes. Tais vetores são calculados, e os componentes de cada vetor são armazenados em componentes de cor do vértice. Esses valores são interpolados no rasterizador e armazenados para cada *pixel* da primitiva rasterizada. A aplicação lê esses valores do *pixel* apontado pelo cursor 2D e utiliza-os para orientar o cursor 3D. A factibilidade desta proposta foi mostrada em [Batagelo and Wu, 2005].

Não é do nosso conhecimento a aplicação da estratégia de codificação de atributos em *pixels* da imagem para movimentos do cursor restritos a características do modelo usando a simulação de um campo de gravidade. Entretanto, sugerimos uma maneira simples de utilizá-lo para este propósito, empregando uma variação do procedimento utilizado para seleção. Em especial, a busca pelo elemento visível mais próximo do cursor 2D pode ser feita no espaço da imagem, verificando os atributos de uma região de *pixels* em torno do *pixel* apontado pelo cursor 2D. Tal procedimento não requer percursos de subdivisões espaciais além daqueles necessários para renderizar a cena 3D. É, portanto, consideravelmente mais simples em comparação com o método de *ray picking*. Por exemplo, para o posicionamento restrito a vértices, atribui-se a cada vértice um identificador único (*e.g.*, uma cor única), além do valor de profundidade utilizado para calcular posteriormente a posição 3D. Somente os vértices do modelo são renderizados, como pontos correspondentes a um *pixel*. Em seguida, a aplicação lê os atributos dos *pixels* contidos em uma área da tela em torno do *pixel* apontado pelo cursor 2D. Essa área define a área de atuação do campo de gravidade, em coordenadas da tela. Para cada *pixel* que contém algum identificador de vértice, a aplicação calcula a distância entre esse *pixel* e o *pixel* apontado pelo cursor 2D. O cursor 2D é então deslocado para o *pixel* mais próximo que contém o identificador, e o cursor 3D é deslocado para a posição 3D correspondente indicada pelos atributos obtidos. Para o posicionamento restrito a arestas, o procedimento é semelhante. Em vez de vértices, o modelo é renderizado em modo aramado, e cada aresta contém um identificador único. Para que a renderização dos vértices ou arestas respeite as relações de oclusão existentes na renderização de primitivas preenchidas, as primitivas preenchidas devem ser renderizadas em um passo anterior no qual apenas o *buffer* de profundidade é alterado. Desse modo, os pontos ou linhas escondidos pelas primitivas preenchidas não serão exibidos.

No caso de posicionamentos restritos de cursor sobre os objetos nas cenas geradas com a técnica de renderização baseada em imagens, as características da imagem poderiam ser determinadas por filtros de pós-processamento de imagem (*e.g.*, um filtro de detecção de bordas), Portanto, a estraté-

gia de codificação de atributos em *pixels* se aplicaria. Para tanto, sugerimos utilizar praticamente o mesmo procedimentos empregado para a restrição a vértices ou arestas. Por exemplo, o modelo pode ser renderizado com as primitivas preenchidas de modo que, em cada *pixel*, seja armazenado o valor de profundidade da superfície naquele ponto. Na imagem efetivamente visualizada, aplica-se o filtro de processamento de imagens desejado (*e.g.*, detecção de bordas). A imagem obtida pode então ser utilizada como uma máscara sobre a imagem no *buffer* de renderização não visível. São excluídos todos os *pixels* que não coincidem com as bordas detectadas. Assim, basta determinar, dentro da região de *pixels* em torno do cursor 2D, qual o *pixel* mais próximo do cursor, considerando apenas aqueles que não foram excluídos pela máscara. O cursor 2D é deslocado para este *pixel*, e o cursor 3D é deslocado para a posição 3D correspondente no modelo.

3.4 Considerações finais

Ao utilizarmos a estratégia de interação baseada na leitura de dados a partir da imagem da cena renderizada em *buffers* de renderização não visíveis, os dados podem ser divididos entre atributos geométricos e atributos não geométricos. Os atributos não geométricos são valores genéricos definidos pela própria aplicação, tais como os valores de identificação de modelos/arestas/vértices utilizados nas tarefas de seleção e posicionamento restrito. Os atributos geométricos vistos até aqui são, além da posição 3D, propriedades de geometria diferencial tais como o vetor normal, vetores tangentes alinhados de acordo com a parametrização das coordenadas de textura da superfície, direções principais, curvaturas principais e derivadas dessas curvaturas.

Em tarefas de seleção, são necessários apenas os atributos armazenados no *pixel* apontado pelo cursor 2D. Para tarefas de posicionamento restrito, esses atributos devem ser armazenados somente para uma região de *pixels* em torno do cursor 2D, região essa determinada pela magnitude do campo de gravidade que está sendo simulado. Estes atributos, relativos à geometria original submetida à GPU, podem sofrer modificações ao longo do fluxo programável de renderização (figura 2.3) e gerar efeitos visuais de profundidade e de curvatura bastante distintos dos efeitos que a geometria original produziria. Uma arquitetura de interação ideal seria aquela que estima na GPU todos os atributos geométricos necessários à interação, de forma transparente à aplicação.

Os atributos geométricos podem ser empregados para aproximar a reconstrução da superfície nos pontos de seleção ou posicionamento restrito, e o uso combinado de atributos geométricos e não geométricos permite a realização de tarefas de interação mais complexas. Por exemplo, para realizar uma tarefa de pintura 3D [Hanrahan and Haeberli, 1990, Carr and Hart, 2004], a arquitetura é capaz de fornecer atributos tais como o identificador do modelo apontado (para identificar qual modelo está sendo pintado), as coordenadas de textura do ponto sob o cursor 2D (para que a aplicação possa

modificar a textura que está sendo pintada) e o vetor normal da superfície nesse ponto (caso a pintura 3D considere o mapeamento do pincel segundo o plano tangente que ele forma sobre esse ponto do modelo).

Neste trabalho de tese, propomos uma arquitetura de suporte a tarefas de interação capaz de estimar, de forma transparente à aplicação, os atributos geométricos necessários ao estágio de interação após o processamento de vértices, levando em consideração a geometria modificada neste estágio (capítulo 4). Todas as modificações de atributos, ocorridas após este estágio e que impactam nas aparências visuais, devem ser tratadas pelo desenvolvedor de aplicativos a fim de que os atributos geométricos sejam devidamente atualizados após o processamento de fragmentos também.

Capítulo 4

Cálculo de elementos de geometria diferencial discreta na GPU

Uma vez que consideramos a possibilidade do uso de técnicas de animação e modelagem geométrica na GPU capazes de deformar a geometria original submetida pela CPU, precisamos levar em conta a presença de tais modificações ao utilizar como atributos de interação as propriedades de geometria diferencial desses modelos. O motivo dessa preocupação torna-se evidente ao observarmos os efeitos do cálculo de iluminação em geometria deformada na GPU. Nas técnicas de mapeamento de detalhes 3D, as bases tangentes (vetor normal, vetor tangente e bitangente) associadas a cada vértice precisam ser atualizadas de acordo com as deformações de modo a realizar cálculos corretos de iluminação. A figura 4.1 ilustra isso numa esfera texturizada com *normal mapping* e iluminada com sombreamento Phong, mas ao mesmo tempo deformada no processador de vértices por uma função de deslocamento da posição de cada vértice ao longo de sua normal, baseada no algoritmo de ruído de Perlin [Perlin, 1985]. No quadro da esquerda, a esfera é iluminada sem que sejam consideradas as bases tangentes atualizadas de acordo com a deformação. Em outras palavras, as bases são as mesmas da esfera não deformada, mostrada no detalhe. Observe que a iluminação neste caso, tanto nos polígonos como fragmentos da esfera após o mapeamento de detalhes, é a mesma da esfera não deformada. As áreas de luz especular não são modificadas e as ondulações na superfície não são percebidas corretamente. No quadro da direita, a esfera é iluminada corretamente considerando as propriedades de geometria diferencial atualizadas após as deformações. Transpondo isso para o contexto de manipulações diretas 3D, é razoável considerar que, da mesma forma que se espera que a iluminação do modelo considere as deformações da geometria na GPU, o usuário também espera que um cursor com movimento restrito à superfície acompanhe a geometria visualizada, *i.e.*, a esfera deformada.

Em geral, a atualização de propriedades tais como vetor normal e vetores tangentes após as de-

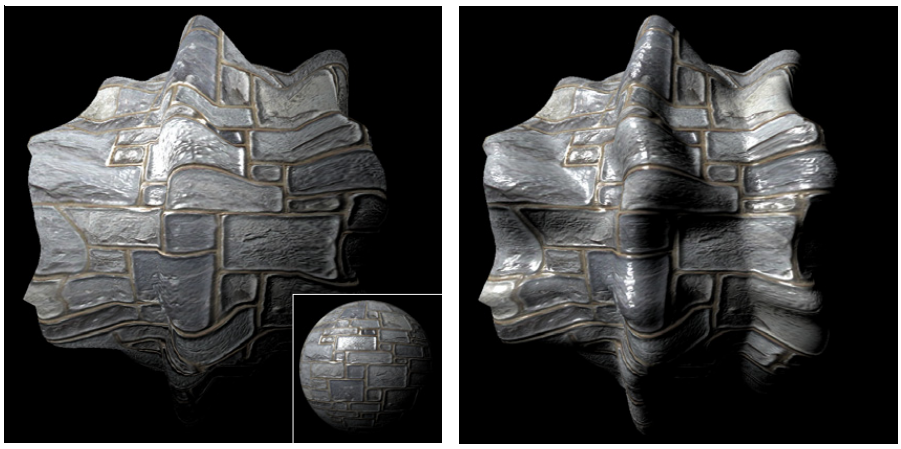


Fig. 4.1: Esfera deformada na GPU por ruído de Perlin, com *normal mapping* e sombreamento Phong. Esquerda: iluminação usando propriedades de geometria diferencial da esfera original (detalhe). Direita: iluminação usando propriedades de geometria diferencial calculadas na GPU após a deformação.

formações não pode ser obtida por uma simples transformação rígida sobre as quantidades originais, uma vez que as deformações geralmente não preservam as propriedades locais de geometria diferencial como área e curvatura. Se o modelo deformado pode ser descrito por uma função paramétrica, as bases tangentes e outros elementos de geometria diferencial podem ser obtidos através das derivadas parciais dessa função. Infelizmente, a condição de existência de tais funções dificilmente é atendida, e uma solução mais geral é necessária. Neste capítulo, mostramos como podemos realizar a estimativa desses elementos de forma eficiente para modelos geométricos deformados arbitrariamente no processador de vértices. Em razão de considerarmos deformações da geometria diretamente na GPU, essa estimativa é feita inteiramente na GPU após as deformações com relação aos vértices.

Para a estimativa de elementos de geometria diferencial de primeira ordem aos vértices, adaptamos à GPU algoritmos tradicionais utilizados na CPU. Em particular, para o cálculo de vetores normais nos baseamos na técnica de Calver [2004]. Para o cálculo de vetores tangentes e bitangentes, utilizamos o algoritmo de Lengyel [2003]. Para o cálculo da estimativa discreta das propriedades de geometria diferencial de segunda ordem ou ordens mais elevadas, propomos um novo algoritmo capaz de produzir resultados robustos e eficientes apenas a partir da vizinhança de "1-anel" de malhas triangulares ou vizinhança imediata de nuvens de pontos[1]. Desse modo, mesmo para modelos compostos de milhares de vértices, é possível estimar em tempo real propriedades tais como os coeficientes

[1]Em malhas triangulares, a vizinhança de *1-anel* de um ponto p é o conjunto de pontos adjacentes a p através de uma única aresta. Em uma nuvem de pontos P, essa vizinhança pode ser interpretada como uma vizinhança imediata $P_d(p)$ dada pelo conjunto de pontos com distância Euclidiana menor ou igual a um valor d: $P_d(p) = \{x \in P \mid |p-x| \leq d\}$ [Tang and Agam, 2005].

da segunda forma fundamental, curvaturas principais, direções principais e derivada das curvaturas. Através da comparação com outras técnicas, também mostramos a robustez de nosso método em malhas amostradas de forma irregular. Tanto quanto na computação de propriedades de primeira ordem, essas propriedades adicionais são calculadas para cada vértice e interpoladas linearmente para cada *pixel* das primitivas rasterizadas.

O restante deste capítulo está organizado da seguinte forma: a seção 4.1 apresenta uma revisão bibliográfica dos métodos mais utilizados para realizar estimativas de propriedades de geometria diferencial, a seção 4.2 apresenta nossas propostas de algoritmos de estimativa de propriedades de geometria diferencial segundo os requisitos necessários para a adequação no contexto da arquitetura de suporte a tarefas de interação 3D, e na seção 4.3 apresentamos resultados de testes de eficiência e robustez de nossos algoritmos em comparação com as técnicas anteriores descritas na seção 4.1.

Para manter uma uniformidade na apresentação de trechos de códigos dos *shaders* neste capítulo, utilizamos apenas código em HLSL (*High-Level Shader Language*) [Microsoft, 2006].

4.1 Estimativas em superfícies discretas

Nesta seção revisamos as principais técnicas existentes na literatura sobre a estimativa de elementos de geometria diferencial em superfícies discretas, particularmente em malhas triangulares.

4.1.1 Elementos de primeira ordem

Os elementos de geometria diferencial de primeira ordem são as quantidades obtidas das primeiras derivadas parciais da equação paramétrica da superfície, tais como o vetor normal, vetores tangentes das curvas coordenadas da superfície e tensor métrico. Na arquitetura proposta utilizamos estimativas de vetor normal e vetores tangentes alinhados de acordo com a parametrização das coordenadas de textura.

Vetor normal

O vetor normal a uma face triangular não degenerada é obtido através do produto vetorial entre quaisquer dois vetores tangentes distintos sobre a face (*e.g.*, dois vetores coincidentes com as arestas do triângulo). Ao considerar que a malha triangular aproxima uma superfície suave, é necessário definir um vetor normal em cada vértice de tal modo que este vetor aproxime o vetor normal da superfície suave em tais pontos.

Se a superfície é descrita por uma função explícita, implícita ou paramétrica, o vetor normal a qualquer vértice pode ser calculado analiticamente. Infelizmente, este não é o caso em aplicações

de Computação Gráfica que utilizam modelos digitalizados ou modelos criados por artistas gráficos. Assim, para um caso geral, os vetores normais devem ser calculados apenas a partir das informações da malha geométrica. Gouraud [1971] foi o primeiro a tratar de tal problema com o objetivo de simular a iluminação de superfícies suaves a partir de malhas poligonais. O vetor normal a um vértice é aproximado através da normalização da soma de todos os vetores normais às faces adjacentes ao vértice. Assim, o vetor normal de cada face contribui igualmente para o cálculo da normal ao vértice, que é uma média dos vetores normais dessas faces.

Thürmer and Wüthrich [1998] propuseram um algoritmo semelhante ao de Gouraud [1971], mas no qual a soma dos vetores normais às faces é ponderada de acordo com o ângulo compreendido entre as duas arestas de cada face, adjacentes ao vértice. Assim, faces com maiores ângulos de incidência terão um peso maior no cálculo da média no vértice. Chen and Schmitt [1992] e Taubin [1995] propuseram métodos parecidos, mas com pesos determinados pela área das faces em vez dos ângulos de incidência. Max [1999] também propôs métodos com ponderação baseada na área das faces, e comparou seus resultados com diferentes tipos de superfícies analíticas de modo a validá-las. Em especial, uma de suas técnicas consiste em ponderar o vetor normal de cada face pelo valor recíproco do produto entre os comprimentos das arestas adjacentes. Os resultados mostram que esta estratégia aproxima localmente a superfície de uma esfera. De fato, em uma esfera discretizada, o algoritmo de Max [1999] produz normais exatas em cada vértice, independentemente do refinamento da geometria.

Praticamente todas as demais técnicas de estimativa do vetor normal em malhas poligonais são baseadas no cálculo da média dos vetores normais das faces incidentes em cada vértice. Apesar dessa variedade de técnicas, não há consenso a respeito de qual delas é a mais efetiva, uma vez que isso depende essencialmente da configuração dos modelos utilizados. Na prática, a técnica de Gouraud [1971] ainda é a mais utilizada em razão de sua simplicidade.

Nas atuais GPUs, o processador de vértices não é capaz de acessar as informações de adjacência de cada vértice processado, nem os atributos dos vértices que participam dessas relações de adjacência. Por outro lado, a estimativa de elementos de geometria diferencial de primeira ordem depende fundamentalmente dessas informações. O cálculo de vetores normais aos vértices requer informações tais como o número de faces compartilhadas por um vértice ou a posição dos três vértices que compõem um triângulo. Para o cálculo de vetores tangentes e bitangentes utilizados em mapeamento de detalhes 3D, também é necessário o acesso às coordenadas de textura de cada vértice da relação de adjacência.

Explorando a funcionalidade de amostrar texturas diretamente no processador de vértices de placas gráficas compatíveis com o modelo de *shader* 3.0 [Microsoft, 2006], Calver [2004] propôs a codificação de dados de adjacência e atributos de vértices como cores de elementos de texturas (*tex-*

els) que são posteriormente lidas no processador de vértices. Com base nesta estratégia, o autor cria um algoritmo para calcular, diretamente na GPU, normais às faces e aos vértices após as deformações de geometria em relação aos vértices. A seguir detalhamos essa estratégia, que é baseada na possibilidade de usar a GPU como um processador de fluxo de propósito geral.

Dados de vértices são tradicionalmente fornecidos ao processador de vértices na forma de um *buffer* de atributos de vértices com acesso apenas de leitura, e no qual o número e tipo de atributos são definidos pela aplicação de acordo com um conjunto de semânticas de entrada definidas pela API. Por exemplo, um conjunto de atributos de vértices de entrada composto por uma posição em coordenadas homogêneas (XYZW), um vetor normal (XYZ) e coordenadas de textura (UV) pode ser descrito pelo código mostrado a seguir:

```
struct VertexIn {
  float4 vPos    : POSITION;
  float3 vNormal : NORMAL;
  float2 vTex    : TEXCOORD0;
};
```

O processador de vértices acessa apenas um conjunto de atributos de vértices de cada vez, que é o conjunto de atributos do vértice que está sendo processado isoladamente em cada momento. Com GPUs compatíveis com o modelo de *shader* 3.0, é possível contornar esta limitação através do acesso, no processador de vértices, de dados de vértices armazenados em mapas de textura, em oposição a dados armazenados em atributos variáveis de entrada [Lefohn, 2004]. Neste caso, texturas são consideradas como arranjos (*arrays*) 2D de dados de propósito geral, e a amostragem dessas texturas é interpretada como operações de acesso aos índices de tais arranjos. Por exemplo, para uma textura RGB em formato de ponto flutuante de 128 bits por *texel*, cada *texel* pode armazenar uma posição XYZ de vértice em 32 bits, codificando cada valor numa componente de cor. Tal textura pode ser utilizada para armazenar a lista de posição dos vértices da geometria, e cada *texel* corresponde então à posição de um vértice. Chamamos esta textura de *mapa de posição de vértices*. O mesmo princípio se aplica a outros atributos de vértices, tais como normais aos vértices (*mapa de normais aos vértices*) e coordenadas de textura (*mapa de coordenadas de textura*). Dessa forma, em vez de utilizar a estrutura de dados tradicional de atributos de vértices para o *buffer* de vértices de entrada do processador de vértices, apenas um atributo é utilizado para cada vértice. Este atributo é o índice unidimensional do vértice no *buffer* de vértices original. A estrutura de atributos de vértices de entrada é simplificada como a seguir:

```
struct VertexIn {
  float iIdx : TEXCOORD0;
```

```
};
```

onde `iIdx` é tal índice. Embora este seja um inteiro, sempre utilizamos números em ponto flutuante uma vez que nem todas as plataformas de *hardware* gráfico têm suporte nativo para inteiros no modelo de *shader* 3.0. De modo a amostrar o *texel* correto do mapa de posição de vértices – e qualquer outro mapa de atributos de vértices – que contém os dados do vértice correspondente ao índice, precisamos converter este índice 1D para um índice 2D, que é o par de coordenadas (u, v) do *texel*. Isto é feito utilizando o seguinte código:

```
float2 Index1DTo2D( float iIdx, float2 vCons ) {
  return float2( iIdx, iIdx ) * vCons;
}
```

onde `vCons` é um vetor de dois componentes pré-computados, armazenado num registrador de constante do *shader*. Esse registrador contém o valor $\langle w_{map}^{-1}, (w_{map} * h_{map})^{-1} \rangle$, onde w_{map} e h_{map} denotam a largura e altura do mapa de atributo de vértices, em *texels*. Esta função é traduzida para uma única instrução `mul` no *assembly* da GPU, de modo que ela é bastante eficiente. Como observado por Calver [2004], a operação de módulo não precisa ser utilizada explicitamente porque ela pode ser simulada pela API através do ajuste do *flag* de modo de endereçamento da textura (`D3DTADDRESS_WRAP` em Direct3D, `GL_TEXTURE_WRAP_S` em OpenGL) sempre que um mapa de textura com atributos de vértices é amostrado.

No trabalho de Calver [2004], dados de adjacência são armazenados de forma semelhante aos dados de atributos de vértices, *i.e.*, usando texturas como arranjos 2D de propósito geral. Os únicos dados de conectividade necessários para calcular normais aos vértices são os índices das faces adjacentes a cada vértice e os índices dos vértices que compõem tais faces. Duas texturas são utilizadas para esse fim: um mapa contendo a lista de faces adjacentes a cada vértice (*mapa de adjacência*), com valência máxima de vértices limitada a oito, e um mapa de índices aos vértices que compõem cada face (*mapa de faces*). O mapa de faces é formatado como uma textura RGB na qual cada *texel* contém os três índices de vértices que compõem uma face. O mapa de adjacência é criado de forma similar, mas neste caso cada grupo de dois *texels* consecutivos corresponde a um dado de adjacência de um único vértice. Em especial, os *texels* são formatados como componentes de cor $RGB\alpha$ que armazenam até oito índices (um para cada componente de cor dos dois *texels*) de faces adjacentes a cada vértice.

Ao utilizar texturas para armazenar arranjos de dados de geometria e dados de adjacência, os processadores de vértices e de fragmentos no modelo de *shader* 3.0 podem acessar dados de quaisquer faces ou vértices por meio de amostragem dessas texturas. Além disso, é possível utilizar a funcionalidade de renderização em texturas de modo a modificar o conteúdo desses arranjos sem a

intervenção da CPU. Isto pode ser feito no processador de fragmentos ao amostrar cada *texel* do mapa de textura de origem, modificar a informação obtida e escrever o resultado a uma textura de destino com um mapeamento biunívoco entre os *texels* de ambas as texturas. Tal procedimento é freqüentemente implementado através da renderização de um quadrilátero que cobre a textura de destino (o alvo de renderização), e mapeado com a textura de origem de modo que cada fragmento rasterizado do quadrilátero corresponde a um *texel* da textura de destino. Conforme detalhamos na seção 2.4.3, isto simula o funcionamento de um processador de fluxo no qual cada elemento de um fluxo de entrada é processado por uma função *kernel* (neste caso, o *shader* de fragmentos) e o resultado é enviado a um fluxo de saída.

Os únicos atributos associados aos vértices do quadrilátero são as coordenadas de textura que mapeiam a textura de origem no quadrilátero. Uma vez que os *texels* da textura de origem são mapeados de forma biunívoca aos *texels* da textura de destino, as coordenadas de textura interpoladas, fornecidas como entrada do processador de fragmentos, correspondem automaticamente aos índices dos arranjos convertidos em 2D. Dessa forma, a função `Index1DTo2D` não precisa ser utilizada neste caso.

No algoritmo de Calver [2004], os vetores normais podem ser calculados simultaneamente com a renderização do modelo, ou podem ser armazenados em novas texturas – evitando, assim, a transferência de dados entre a CPU e a GPU – que podem ser amostradas no processador de vértices durante o passo de renderização. Como forma de simplificar a implementação, o algoritmo impõe um limite máximo de 6 a 8 faces adjacentes a cada vértice (o vetor normal ao vértice é calculado como a média dos vetores normais das faces adjacentes).

O algoritmo é baseado no algoritmo de Blinn [1977] que primeiramente calcula os vetores normais às faces e, para cada vértice, normaliza os vetores normais acumulados das faces adjacentes. Na GPU, isso é implementado como três passos de renderização usando a estratégia de processamento de fluxo descrita anteriormente. O primeiro passo armazena os atributos modificados da geometria deformada, enquanto que os outros dois utilizam estes dados para finalmente calcular as normais às faces e aos vértices:

1. **Atualização da posição dos vértices**: Cada *texel* do mapa de posição de vértices é lido e processado pela função de deformação dependente da aplicação. O resultado é armazenado em um novo mapa de posição de vértices que é utilizado nos passos subseqüentes.

2. **Cálculo das normais às faces**: Cada *texel* do mapa de faces é lido de modo a obter os 3 índices dos vértices que compõem a face. O mapa de vértices atualizado é então amostrado nos índices dados de modo a obter a posição 3D de cada vértice da face, no sistema de coordenadas do objeto. Com tais dados, a normal à face é obtida através do produto vetorial de quaisquer dois

vetores formados das arestas dos triângulos, começando de um vértice comum. O resultado é armazenado em um mapa de normais às faces.

3. **Cálculo das normais aos vértices**: Para cada vértice, é amostrado um grupo de dois *texels* subseqüentes no mapa de adjacência. Isso corresponde à leitura dos índices de até oito faces adjacentes ao vértice, um para cada componente de cor $RGB\alpha$ dos dois *texels*. Para cada um desses índices de faces, o mapa de normais às faces é amostrado de modo a obter a normal à face correspondente. As oito normais às faces são somadas e o resultado é normalizado para obter a normal ao vértice. Tal resultado é armazenado em um mapa de normais aos vértices.

Com o mapa de normais aos vértices calculado, o *shader* de vértices pode acessar seus dados de modo a fornecer ao *shader* de fragmentos as normais corretas para realizar os cálculos de iluminação. Uma vez que os resultados são armazenados no mapa de normais aos vértices, eles só precisam ser calculados novamente caso a geometria seja afetada por uma nova deformação.

Vetores tangente e bitangente

Segundo Lengyel [2003], vetores tangentes alinhados de acordo com o mapeamento das coordenadas de textura podem ser obtidos através do cálculo, para cada face, de vetores T (tangente) e B (bitangente) tais que qualquer ponto 3D Q dentro de um triângulo com vértices 3D P_n $(n = 0, 1, 2)$ e coordenadas de textura (s_n, t_n) pode ser representado por:

$$Q - P_n = (Q_s - s_n)T + (Q_t - t_n)B, \tag{4.1}$$

onde (Q_s, Q_t) são as coordenadas de textura associadas a Q.

Atribuindo P_0 a P_n e substituindo Q por P_1 e P_2, obtemos o seguinte sistema de equações lineares:

$$\begin{aligned} P_1 - P_0 &= (s_1 - s_0)T + (t_1 - t_0)B, \\ P_2 - P_0 &= (s_2 - s_0)T + (t_2 - t_0)B. \end{aligned} \tag{4.2}$$

Resolvendo para $\mathbf{T}$ e $\mathbf{B}$, temos:

$$\begin{aligned} T &= ((P_1 - P_0)t_2 - (P_2 - P_0)t_1)c, \\ B &= ((P_2 - P_0)s_1 - (P_1 - P_0)s_2)c, \end{aligned} \tag{4.3}$$

onde $c = (s_1 t_2 - s_2 t_1)^{-1}$.

Com T e B calculados, e considerando o vetor normal N no espaço do objeto correspondente, ortogonalizamos esses vetores de modo a produzir a base tangente $\{T, B, N\}$. Em mapeamento de

detalhes 3D, essa base é utilizada para transformar o vetor de direção da fonte de luz L_{os} do espaço do objeto ao vetor L_{ts} no espaço tangente, usando a seguinte matriz:

$$L_{ts} = \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{bmatrix} L_{os}. \tag{4.4}$$

A operação de ortogonalização é feita usando Gram-Schmidt em T com relação a N:

$$T' = T - (N \cdot T)N. \tag{4.5}$$

O vetor bitangente B é então obtido do produto vetorial entre T' e N. Entretanto, precisamos considerar que a regra de mão utilizada em cada face ($N \times T$ ou $T \times N$) não é conhecida e pode variar entre as faces. Isso ocorre, por exemplo, quando uma textura é mapeada de forma espelhada sobre a superfície. Este é um procedimento comum em aplicativos de jogos, para diminuir o consumo de memória de vídeo. Para manter a consistência de que o vetor normal da superfície deve apontar na mesma direção do vetor normal no espaço tangente (*e.g.*, para fora do modelo), a regra de mão deverá ser invertida nas faces com mapeamento espelhado. Para isso, o vetor B deverá ser calculado da forma indicada na equação 4.3, e não simplesmente através do produto vetorial entre N e T.

Assim como as normais aos vértices são calculadas a partir da suposição de que a malha triangular aproxima uma superfície suave, assim também as bases tangentes são calculadas: inicialmente nas faces e depois ponderadas e convertidas em bases ortonormais nos vértices.

4.1.2 Elementos de segunda ordem

A estimativa de elementos de geometria diferencial de segunda ordem em superfícies discretas tem sido estudada extensivamente nos últimos anos em razão do surgimento de um grande número de aplicações que fazem uso destes elementos (*e.g.*, reamostragem, filtragem, renderização não fotorealista, interação). Diversas abordagens para solução deste problema foram apresentadas e se distinguem entre si especialmente com relação à robustez da estimativa ante diferentes configurações da malha da geometria, mas também pela simplicidade de implementação e desempenho. Vamos nos concentrar aqui nas propostas utilizadas para estimar o tensor de curvatura $\mathbb{III}_s$ e assim determinar as curvaturas e direções principais.

Todas as técnicas de estimativa do tensor de curvatura que analisamos consideram que as curvas coordenadas de S são ortogonais, de modo que $\mathbb{III}_s$ é a própria matriz de Weingarten. Através do acesso a uma região de vértices, arestas ou faces adjacentes a cada vértice da geometria, obtém-se uma estimativa de $\mathbb{III}_s$ para cada vértice. A estimativa correspondente para cada ponto da superfície é

obtida através da interpolação linear dos tensores calculados nos vértices.

Na aproximação do tensor de curvatura, supõe-se em geral a existência prévia de uma estimativa do vetor normal ao vértice. Essa estimativa é normalmente obtida pela média dos vetores normais das faces adjacentes ao vértice. As diversas técnicas de estimativa do vetor normal apresentadas na literatura se diferem pelo modo como essa média é ponderada, como visto na seção 4.1.1. Não obstante, o método mais difundido consiste simplesmente em normalizar a soma dos vetores (não unitários) normais às faces adjacentes [Gouraud, 1971].

Infelizmente, não há consenso sobre qual abordagem produz resultados mais acurados para estimar o vetor normal de vértices de uma malha triangular arbitrária, uma vez que essa malha pode representar diferentes tipos de superfícies suaves. Por outro lado, conforme observado por Goldfeather and Interrante [2004], mesmo quando uma malha é amostrada sobre uma superfície analítica e tem vetores normais exatos em cada vértice, há outras fontes de introdução de erro na estimativa do tensor de curvatura, como a não regularidade da amostragem. De modo a comparar com nossa proposta, as técnicas apresentadas a seguir são divididas nas seguintes classes: (1) aproximação por superfícies analíticas; (2) aproximação pela curvatura normal; (3) aproximação pela média do tensor de curvatura.

Aproximação por superfícies analíticas

O método mais tradicional de estimar propriedades de geometria diferencial de uma malha triangular é através da aproximação de uma superfície paramétrica a uma região local de pontos da malha. A matriz de Weingarten pode então ser obtida dessa superfície de forma analítica [Hamann, 1993, Cazals and Pouget, 2003, Goldfeather and Interrante, 2004].

Inicialmente são escolhidos dois vetores ortonormais B e T no plano tangente em p com vetor normal N de modo a formar uma base (B, T, N) em $\mathbb{R}^3$. Cada vértice q adjacente a p é transformado nessas coordenadas locais de modo que $p = (0, 0, 0)$ e $N = (0, 0, 1)$. Uma superfície quadrática nesse sistema terá a seguinte forma:

$$z = f(x, y) = \frac{A}{2}x^2 + Hxy + \frac{C}{2}y^2, \tag{4.6}$$

onde os coeficientes A, H e C são os próprios coeficientes da matriz de Weingarten: e, f, g. Utilizando a informação de cada vértice adjacente é possível construir o sistema de equações:

$$\begin{bmatrix} \frac{1}{2}x_i^2 & x_i y_i & \frac{1}{2}y_i^2 \end{bmatrix} \begin{bmatrix} e \\ f \\ g \end{bmatrix} = z_i \qquad \text{onde} \qquad i = 1, ..., n, \tag{4.7}$$

cuja solução pode ser encontrada pelo método de mínimos quadrados. Goldfeather and Interrante [2004] observam que, para comparar esse sistema de equações com aqueles obtidos nas abordagens de aproximação por curvatura normal e derivada direcional da normal, cada i-ésima equação pode ser multiplicada pelo fator $\frac{2}{k_i^2}$, onde $k_i = \sqrt{x_i^2 + y_i^2}$. Fazendo $(x_i, y_i) = k_i(u_i, v_i)$, obtém-se:

$$\begin{bmatrix} \frac{1}{2}x_i^2 & x_i y_i & \frac{1}{2}y_i^2 \end{bmatrix} \begin{bmatrix} e \\ f \\ g \end{bmatrix} = \frac{k_i^2}{2} \begin{bmatrix} u_i^2 & 2u_i v_i & v_i^2 \end{bmatrix}, \tag{4.8}$$

$$\begin{bmatrix} u_i^2 & 2u_i v_i & v_i^2 \end{bmatrix} \begin{bmatrix} e \\ f \\ g \end{bmatrix} = d_i, \tag{4.9}$$

onde $d_i = \frac{2}{k_i^2} z_i$. Ao isolar z_i, obtém-se a equação $z_i = \frac{d_i}{2} k_i^2$ que corresponde à equação da parábola que passa por p (a origem) e pelo ponto (k_i, z_i). Como $k_i = \sqrt{x_i^2 + y_i^2}$, a parábola passa pela origem e pelo ponto (x_i, y_i, z_i) do $\mathbb{R}^3$. Esse resultado indica que a robustez das estimativas do tensor $\mathbb{III}_s$ usando este método depende do quanto a região local em torno de p é melhor aproximada por parábolas nas direções das arestas aos vértices adjacentes. Erros de aproximação são maiores em malhas amostradas de forma irregular, pois nesses casos tende a ser igualmente maior o número de superfícies analíticas que se encaixam simultaneamente numa mesma vizinhança de pontos.

A abordagem de aproximação por superfícies quadráticas pode ser estendida para superfícies de ordem mais elevada. Goldfeather and Interrante [2004] utiliza superfícies cúbicas e combina as informações já existentes do vetor normal aos vértices adjacentes de modo a obter estimativas mais robustas das direções principais. A idéia de aproveitar o vetor normal dos vértices adjacentes foi seguida por outras técnicas baseadas na vizinhança de 1-anel, como a de Rusinkiewicz [2004], e tem se revelado importante para a obtenção de resultados robustos em malhas irregulares.

Aproximação pela curvatura normal

Conforme foi mostrado na seção 3.1.2, a multiplicação de $\mathbb{III}_s$ por um vetor U duas vezes produz um escalar que corresponde à curvatura normal na direção U:

$$\mathbb{III}_s(U, U) = U' \mathbb{III}_s U = \kappa_U. \tag{4.10}$$

Para superfícies discretas, uma estimativa da curvatura normal na direção de uma aresta entre um vértice p a q pode ser obtida pela fórmula da curvatura do círculo osculador que passa por esses

vértices [Chen and Schmitt, 1992, Meyer et al., 2003]:

$$\kappa_{pq} = 2\frac{(p-q)\cdot N_p}{\|p-q\|^2}, \tag{4.11}$$

onde N_p é o vetor normal em p. Combinando as equações 4.10 e 4.11 e fazendo $U = (p-q) = \begin{bmatrix} u_i & v_i \end{bmatrix}$, é possível construir o seguinte sistema de n equações, onde n é o número de arestas adjacentes a p:

$$U_i' \mathbb{III}_s U_i = \kappa_{U_i} \quad \text{onde} \quad i = 1, 2, ..., n, \tag{4.12}$$

$$\begin{bmatrix} u_i & v_i \end{bmatrix} \begin{bmatrix} e & f \\ f & g \end{bmatrix} \begin{bmatrix} u_i \\ v_i \end{bmatrix} = \kappa_{U_i}, \tag{4.13}$$

$$\begin{bmatrix} u_i^2 & 2u_iv_i & v_i^2 \end{bmatrix} \begin{bmatrix} e \\ f \\ g \end{bmatrix} = \kappa_{U_i}. \tag{4.14}$$

Este sistema pode ser resolvido para $\mathbb{III}_s$ usando mínimos quadrados [Chen and Schmitt, 1992]. Comparando este conjunto de equações com aquele obtido pela aproximação de superfícies quadráticas, verifica-se que os dois diferem apenas no tipo de curvas aproximadas na direção de cada aresta: parábolas, no caso da aproximação por superfícies quadráticas; círculos, na aproximação pela curvatura normal. Dessa forma, a aproximação pela curvatura normal possui as mesmas deficiências da aproximação por superfícies analíticas. Rusinkiewicz [2004] destaca como principal deficiência dessas técnicas a ambigüidade no cálculo da curvatura em configurações de geometria com vértices coincidentes com a interseção de duas linhas. Em tais regiões, tanto uma superfície plana como hiperbólica se ajustam ao conjunto de pontos.

Aproximação pela média do tensor de curvatura

Esta abordagem é baseada na possibilidade de estimar um tensor de curvatura para cada aresta e realizar uma média entre os tensores calculados para todas as arestas contidas em uma região da malha em torno de um vértice p. Considera-se que, para cada aresta E de uma malha triangular, há uma curvatura mínima e máxima associada, sendo que a curvatura mínima encontra-se na direção da aresta e a curvatura máxima na direção perpendicular que cruza a aresta ao longo do plano tangente. Tal condição permite definir um tensor para cada ponto da aresta. A média entre os tensores calculados para as arestas dentro de uma região em torno de p é dada por:

$$\mathbb{T}_s(p) = \frac{1}{|A|} \sum_{\text{arestas } U} \beta(U)\, |U \cap A|\, \bar{U}\, \bar{U}', \tag{4.15}$$

onde p é um vértice da malha, $|A|$ é a área da superfície discreta em torno de p e sobre a qual o tensor está sendo estimado, $\beta(U)$ é o ângulo (positivo se convexo, negativo se côncavo) entre os vetores normais dos dois triângulos que compartilham a aresta U, $|U \cap A|$ é o comprimento de $U \cap B$ (entre 0 e $|U|$), e $\bar{U}$ é um vetor (coluna) unitário paralelo à aresta U.

$\mathbb{T}_s$ difere de $\mathbb{II}_s$ por ser um tensor 3×3 que inclui informações sobre o vetor normal. Em especial, o autovetor associado ao autovalor de menor magnitude é uma estimativa do vetor normal em p. Os dois outros autovalores correspondem a $\kappa_{min}, \kappa_{max}$, mas com autovetores trocados, *i.e.*, o autovetor associado ao autovalor de κ_{min} indica a direção da curvatura máxima, e vice-versa.

As técnicas de aproximação pela média do tensor de curvatura não sofrem das mesmas deficiências das técnicas de aproximação por superfícies analíticas e por curvatura normal em vértices coincidentes com a interseção de duas linhas. Entretanto, outras configurações podem acumular erros que não são minimizados mesmo quando se aumenta o número de arestas utilizadas na avaliação da equação 4.15. Uma dessas configurações é destacada por Rusinkiewicz [2004]: os pólos de uma esfera geodésica construída a partir da subdivisão de um tetraedro.

Procurando resolver as deficiências dessas técnicas [Cohen-Steiner and Morvan, 2003, Alliez et al., 2003], Rusinkiewicz [2004] propõe um novo método de aproximação pela média do tensor de curvatura, baseando-se na idéia de Goldfeather and Interrante [2004] de utilizar o vetor normal já existente nos vértices da vizinhança de 1-anel. De forma semelhante aos algoritmos tradicionais de cálculo do vetor normal, o tensor $\mathbb{II}_s$ é calculado inicialmente para as faces e então estimado para os vértices como uma média dos tensores das faces adjacentes. Os resultados mais robustos são obtidos quando essa média é ponderada de acordo com a área da região de Voronoi em torno do vértice de interesse.

O método de Rusinkiewicz [2004] produz resultados acurados tanto para as configurações onde falham os métodos de aproximação por superfícies analíticas (*e.g.*, vértices de interseções entre duas linhas), quanto para as configurações onde falham os outros métodos por média do tensor de curvatura (*e.g.*, pólos de esferas geodésicas). Como conseqüência, o método também produz resultados visualmente mais satisfatórios em modelos digitalizados. Nestes modelos, é comum a presença, na malha triangular, de configurações problemáticas às outras técnicas, o que gera o aparecimento de *outliers* (estimativas grosseiras, porém pontuais) em regiões do modelo que deveriam ter curvatura suave.

Dos algoritmos citados, o método de Rusinkiewicz [2004] é o único a calcular também elementos

de derivadas de terceira ordem, em especial o tensor de derivada de curvatura. Da mesma forma que o tensor de curvatura é obtido através da média dos tensores das faces, os coeficientes do tensor de derivada de curvatura são estimados para cada face e então aproximados para os vértices através de uma média ponderada. A formulação utilizada para estimar os coeficientes do tensor de derivada de curvatura é aquela das equações 3.26 até 3.29, *i.e.*, utilizando as direções principais como bases tangentes locais.

4.2 Nossa proposta

A seguir apresentamos nossa abordagem para a estimativa de propriedades de geometria diferencial na GPU sobre modelos 3D que sofrem deformações de geometria no processador de vértices. A estimativa de elementos de primeira ordem é realizada de acordo com algoritmos já conhecidos na literatura, mas adaptados à arquitetura das atuais GPUs. Para a estimativa de elementos de segunda e terceira ordem, propomos uma abordagem original de fácil implementação em *hardware* gráfico. Através de uma comparação com outras técnicas, mostramos que ela é ao mesmo tempo eficiente e capaz de fornecer resultados robustos até mesmo em malhas amostradas irregularmente ou com ruído de deslocamento ao longo do vetor normal.

4.2.1 Elementos de primeira ordem

Para a estimativa de elementos de primeira ordem, baseamo-nos no método de Calver [2004] que calcula a estimativa do vetor normal em cada vértice da malha segundo a adaptação para a GPU do tradicional algoritmo de dois passos de Blinn [1977]. Entretanto, para tornar o algoritmo mais robusto, removemos a limitação do número faces adjacentes. Ao contrário, consideramos todas as faces adjacentes a cada vértice ao computar a média dos vetores normais nos vértices. Desse modo, os resultados que podemos obter são tão robustos quanto se eles tivessem sido realizados usando o algoritmo de Blinn [1977] na CPU.

Para a proposta de uma arquitetura de suporte a tarefas de interação 3D, estendemos o método de Calver [2004] de modo a calcular os vetores tangentes e bitangentes alinhados segundo a parametrização das coordenadas de textura no modelo [Lengyel, 2003]. Para isso utilizamos o mesmo princípio de armazenar atributos de vértices e relações de adjacência em texturas. Assim como no cálculo de vetores normais, não impomos uma restrição quanto ao número máximo de faces adjacentes a cada vértice.

Para criar um mapa de adjacência no qual o número de faces adjacentes a cada vértice não é fixo, primeiramente calculamos na CPU os índices das faces adjacentes a cada vértice. O resultado é

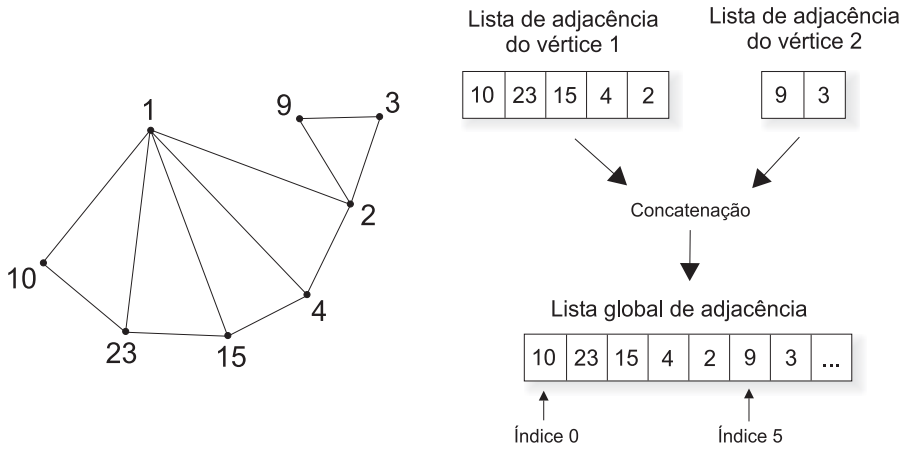


Fig. 4.2: Construção da lista global de adjacência.

armazenado em listas temporárias de adjacência com base nos vértices. Essas listas são então concatenadas em uma única lista global, ordenadas pelo índice do vértice. Por exemplo, se o vértice de índice 1 tem a lista de adjacência com índices $\langle 10, 23, 15, 4, 2\rangle$ e o vértice de índice 2 tem a lista com índices $\langle 9, 3\rangle$, a lista global de adjacência resultante conterá a seqüência inicial $\langle 10, 23, 15, 4, 2, 9, 3, ...\rangle$. Esta lista é então armazenada em um mapa de adjacência no qual cada *texel* contém um único valor da lista global de adjacência (figura 4.2). Tais informações podem ser armazenadas em uma textura com um único componente de cor por *texel*.

De modo a acessar o mapa de adjacência na GPU, precisamos fornecer atributos adicionais para cada vértice da geometria. Em especial, cada vértice deve possuir um atributo que indique o índice da primeira face adjacente na lista de adjacência, e outro atributo que defina o número de faces adjacentes. Isto é feito criando outro mapa de atributos de vértices – o *mapa de índices de adjacência* – no qual cada *texel* corresponde a um vértice e contém dois valores. O primeiro valor é o índice 1D da primeira face adjacente na lista global de adjacência. Para o último exemplo utilizado (figura 4.2), esses valores seriam 0 e 5 para o primeiro e segundo vértices, respectivamente. No *shader* que estima as propriedades de geometria diferencial de primeira ordem, este valor é utilizado para determinar as coordenadas de textura do primeiro *texel* a ser lido do mapa de adjacência. O segundo valor de cada *texel* do mapa de índices de adjacência número de faces adjacentes ao vértice.

O cálculo das bases tangentes é integrado aos *shaders* de fragmentos utilizados para calcular os vetores normais. A seguir detalhamos cada passo de renderização do algoritmo de estimativa de propriedades de geometria diferencial de primeira ordem utilizado em nossa abordagem.

1. **Atualização das posições dos vértices**: A posição original do vértice 3D é lida do mapa de posição dos vértices, processada pela função de deformação dependente da aplicação e escrita no mapa de posições atualizadas dos vértices. Na verdade, esta etapa não faz parte do algoritmo

de estimativas de elementos de primeira ordem em si. Por outro lado, é necessária para o correto funcionamento do algoritmo caso o modelo sofra algum tipo de deformação na GPU. Se essa deformação não existir, o mapa de posições atualizadas não precisa ser empregado, pois o mapa de posições da geometria original já conterá os atributos corretos.

2. **Cálculo das bases tangentes às faces**: Normais às faces também são calculadas como no algoritmo de Calver [2004]. Os índices dos vértices que compõem cada face são lidos do mapa de faces e a posição 3D de cada vértice é lida do mapa de posições atualizadas de vértices. O vetor normal à face (não normalizado) resultante é armazenado no mapa de normais às faces.

   Para calcular os vetores tangentes, é necessário acessar coordenadas de textura para cada vértice da face. Isto é feito através da amostragem de um mapa de atributos de vértices similar ao mapa de posição de vértices, mas contendo coordenadas de textura. Os vetores tangente e bitangente não normalizados são calculados através da solução da equação 4.3. O resultado é armazenado em um mapa de vetores tangentes às faces e um mapa de vetores bitangentes às faces.

   Uma vez que essas operações são executadas no mesmo *shader*, utilizamos a funcionalidade de renderização em múltiplos alvos de renderização para escrever em três diferentes mapas de texturas (mapa de normais às faces, mapa de vetores tangentes às faces e mapa de vetores bitangentes às faces) ao mesmo tempo.

3. **Cálculo das bases tangentes aos vértices**: Para cada vértice, o mapa de índices de adjacência é amostrado de modo a obter o índice da primeira face adjacente na lista de adjacência e o número de faces adjacentes. A seguir, chamamos tais dados de `iFirstAdjFace` e `iNumAdjFace`, respectivamente. Para cada face adjacente n ($n \in [0, \texttt{iNumAdjFace})$), o mapa de adjacência é amostrado com relação ao índice $n + \texttt{iFirstAdjFace}$ de modo a obter o índice da face adjacente que está sendo processada. Os vetores normal, tangente e bitangente à face são obtidos dos mapas de normais às faces, vetores tangentes às faces e vetores bitangentes às faces. A soma das normais às faces é normalizada e o resultado é armazenado em um mapa de normais aos vértices. A soma dos vetores tangentes às faces é ortogonalizada com relação às normais às faces acumuladas usando o algoritmo de Gram-Schmidt. O resultado é normalizado e armazenado em um mapa de vetores tangentes aos vértices. Os vetores bitangentes aos vértices não são armazenados em uma textura, pois podem ser obtidos do produto vetorial entre os vetores normais e os vetores tangentes. Por outro lado, os vetores bitangentes são utilizados para se determinar se, nesses produtos vetoriais, eles apontam na direção obtida pela regra da mão direita ou esquerda. Isso é feito com o produto vetorial $\mathbf{N} \times \mathbf{T}$, comparando a sua direção com a direção do vetor bitangente acumulado. Apenas este resultado é guardado, o que requer um *bit* por *texel* (*e.g.*, 0 para $\mathbf{N} \times \mathbf{T}$, 1 para $\mathbf{T} \times \mathbf{N}$). Na prática, armazenamos esta informação no

canal alfa do mapa de tangentes aos vértices.

Após executar estes três passos de renderização, o *shader* de vértices está agora apto a acessar o mapa de normais aos vértices e os mapa de vetores tangentes para fornecer ao *shader* de fragmentos as bases tangentes corretas da geometria deformada. O vetor bitangente B no vértice é calculado através do produto vetorial entre a normal N e o vetor tangente T, na ordem determinada pela regra de mão (direita ou esquerda): $B = (N \times T)(-1)^h$, onde h indica a regra utilizada (0 para a mão direita, 1 para a mão esquerda).

4.2.2 Elementos de segunda e terceira ordem

Tanto quanto foi possível propor o cálculo de vetores normais, tangentes e bitangentes diretamente na GPU, também propomos calcular diretamente na GPU o tensor de curvatura $\mathbb{II}_s$, as curvaturas principais, as direções principais e o tensor de derivada da curvatura $\mathbb{C}_s$, para cada vértice da geometria deformada. De modo a avaliar a escolha de uma das técnicas apresentadas anteriormente para tal propósito, procuramos priorizar as seguintes características:

- **Eficiência**: O tempo total da estimativa do tensor de curvatura, das curvaturas principais, das direções principais e do tensor de derivada de curvatura deve ser tal que permita a execução do algoritmo em tempo real para modelos compostos de dezenas de milhares de vértices. Como estamos considerando a implementação do algoritmo na GPU, as instruções e estruturas de dados envolvidas devem ser simples o suficiente para permitir essa implementação.

- **Generalidade**: De modo a manter a compatibilidade com malhas triangulares arbitrárias, o algoritmo não deve impor restrições quanto à topologia do modelo. Em especial, o algoritmo deve calcular as propriedades de geometria diferencial mesmo em superfícies com buracos, superfícies amostradas de forma irregular, e malhas com diferentes valências dos vértices. Para o tratamento de gráficos baseados em pontos, deve-se ainda considerar a inexistência de relações de orientação das faces. Em outras palavras, a única estrutura de conectividade permitida deve ser a vizinhança de 1-anel de cada vértice, na forma de um conjunto de pontos possivelmente sem ordem definida.

- **Robustez**: O algoritmo deve calcular corretamente as propriedades de geometria diferencial para as configurações consideradas degeneradas para os métodos citados anteriormente: pontos colineares e pontos de interseção entre duas linhas. Dessa forma, em modelos digitalizados, a presença de *outliers* deve ser minimizada.

Infelizmente, nenhuma das técnicas analisadas preenche simultaneamente todos os requisitos apresentados. Embora as técnicas baseadas na aproximação por superfícies analíticas ou curvatura normal sejam adequadas para a arquitetura das atuais GPU e, portanto, sejam bastante eficientes, não produzem resultados robustos para malhas arbitrárias. Por outro lado, as técnicas de aproximação pela média pelo tensor de curvatura são robustas, mas difíceis de serem implementadas nas GPUs em razão do uso de estruturas de dados complexas ou necessidade de vários passos de renderização. Por exemplo, a técnica de Alliez et al. [2003], baseada na equação 4.15, requer um pré-processamento custoso para determinar quais arestas (e em qual proporção) estão dentro da região B em torno de cada vértice. De forma semelhante, segundo o algoritmo de Rusinkiewicz [2004], a média entre os tensores das faces requer o cálculo da área de Voronoi de cada face que compartilha o vértice que está sendo calculado. Esse cálculo requer pelo menos um passo adicional de renderização. Como o algoritmo utiliza o mesmo princípio do cálculo de vetores normais (um passo para calcular sobre as faces e outro passo para fazer a média), um total de (no mínimo) três passos de renderização para calcular apenas o tensor de curvatura poderia comprometer o desempenho.

De modo a atender os requisitos necessários, propomos uma nova técnica capaz de unir a robustez dos algoritmos de média pelo tensor de curvatura, e a simplicidade e eficiência dos métodos baseados na aproximação por superfícies analíticas ou por curvatura normal. Nossa proposta é uma variação do algoritmo de Rusinkiewicz [2004], tanto para calcular o tensor de curvatura como o tensor de derivada da curvatura [Batagelo and Wu, 2007b].

Estimativa do tensor de curvatura

Para cada vértice, estimamos $\mathbb{II}_s$ a partir da relação entre este tensor e a derivada da curvatura normal numa dada direção no plano tangente em um ponto p:

$$\mathbb{III}_s(U) = \mathbb{III}_s U = D_U N. \tag{4.16}$$

Numa superfície suave, o vetor U pode assumir qualquer direção perpendicular ao vetor normal em p. Em uma malha triangular ou nuvem de pontos, essas direções podem ser aproximadas por diferenças finitas como vetores compreendidos entre o vértice p e cada vértice da vizinhança de 1-anel. De forma semelhante, as derivadas direcionais dos vetores normais podem ser aproximadas pela diferença dos vetores normais previamente estimados nos vértices. As diferenças tanto de posições como de vetores normais devem ser expressas em coordenadas do plano tangente em p. Assim, é possível construir o seguinte sistema de equações lineares para cada vértice:

$$\mathbb{II}_s \begin{bmatrix} (q_i - p) \cdot U \\ (q_i - p) \cdot V \end{bmatrix} = \begin{bmatrix} (N_{q_i} - N_p) \cdot U \\ (N_{q_i} - N_p) \cdot V \end{bmatrix}, \tag{4.17}$$

$$\begin{bmatrix} (q_i - p) \cdot U & (q_i - p) \cdot V & 0 \\ 0 & (q_i - p) \cdot U & (q_i - p) \cdot V \end{bmatrix} \begin{bmatrix} e \\ f \\ g \end{bmatrix} = \begin{bmatrix} (N_{q_i} - N_p) \cdot U \\ (N_{q_i} - N_p) \cdot V \end{bmatrix}, \tag{4.18}$$

onde q_i é um ponto da vizinhança imediata de p ($i = 1, 2, ..., n$, sendo n o número de pontos nessa vizinhança) e U, V são dois vetores ortonormais no plano tangente. Esse sistema pode ser resolvido para $\mathbb{II}_s$ usando mínimos quadrados.

Nossa proposta é similar ao método de Rusinkiewicz [2004], diferindo apenas pelo fato de realizarmos o cálculo diretamente para os vértices, ao invés de realizar um passo para determinar o tensor de curvatura para cada face adjacente ao vértice, e um passo adicional para realizar a média desses tensores. Na GPU, a estimativa dos tensores diretamente para os vértices pode ser realizada em apenas um passo de renderização, aumentando a eficiência.

Estimativa do tensor de derivada da curvatura

Para calcular o tensor de derivada de curvatura, recorremos às equações 3.26 até 3.29 que nos fornecem uma forma simples de obter $\mathbb{C}_s$ em coordenadas relativas às direções principais no plano tangente. Para tanto, primeiro calculamos $\mathbb{II}_s$ e então extraímos seus autovalores (curvaturas principais) e autovetores (direções principais). As direções principais se tornam os novos vetores de base U, V em p, e podemos expressar cada direção entre p e q_i nessas novas coordenadas. Obtemos então o seguinte sistema que expressa, por diferenças finitas, a relação $\mathbb{C}_s(U) = D_U \mathbb{II}_s$ em coordenadas principais:

$$\mathbb{C}_s \begin{bmatrix} \Delta E_i \cdot U \\ \Delta E_i \cdot V \end{bmatrix} = \begin{bmatrix} \Delta K_{min_i} \cdot U & \Delta K_{max_i} \cdot U \\ \Delta K_{min_i} \cdot V & \Delta K_{max_i} \cdot V \end{bmatrix} \quad \text{onde} \quad i = 1, 2, ..., n, \tag{4.19}$$

onde

$$\Delta E_i = q_i - p, \tag{4.20}$$

$$\Delta K_{min_i} = \kappa_{min_{q_i}} - \kappa_{min_p}, \tag{4.21}$$

$$\Delta K_{max_i} = \kappa_{max_{q_i}} - \kappa_{max_p}. \tag{4.22}$$

Expressando $\mathbb{C}_s$ em sua forma vetorial (Eq. 3.19), temos:

$$\left[\begin{bmatrix} a & b \\ b & c \end{bmatrix}\begin{bmatrix} b & c \\ c & d \end{bmatrix}\right]\begin{bmatrix} \Delta E_i \cdot U \\ \Delta E_i \cdot V \end{bmatrix} = \begin{bmatrix} \Delta K_{min_i} \cdot U & \Delta K_{max_i} \cdot U \\ \Delta K_{min_i} \cdot V & \Delta K_{max_i} \cdot V \end{bmatrix}, \tag{4.23}$$

$$\begin{bmatrix} \Delta E_i \cdot U & \Delta E_i \cdot V & 0 & 0 \\ 0 & \Delta E_i \cdot U & \Delta E_i \cdot V & 0 \\ 0 & \Delta E_i \cdot U & \Delta E_i \cdot V & 0 \\ 0 & 0 & \Delta E_i \cdot U & \Delta E_i \cdot V \end{bmatrix}\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} \Delta K_{min_i} \cdot U \\ \Delta K_{min_i} \cdot V \\ \Delta K_{max_i} \cdot U \\ \Delta K_{max_i} \cdot V \end{bmatrix}, \tag{4.24}$$

onde a, b, c, d são os coeficientes de $\mathbb{C}_s$.

Da mesma forma que na estimativa de $\mathbb{II}_s$, este sistema também é resolvido pelo método de mínimos quadrados.

4.3 Resultados

Na GPU, o algoritmo foi implementado como dois *shaders* em HLSL compatíveis com o modelo de *shader* 3.0 e listados no apêndice B. O primeiro *shader* foi utilizado para estimar os coeficientes do tensor de curvatura, as curvaturas principais e as direções principais com base nas equações 4.16, 4.17 e 4.18. O segundo *shader* foi utilizado para estimar os coeficientes do tensor de derivada de curvatura através das equações 4.19 a 4.24. Este segundo *shader* requer a execução do primeiro *shader* de modo a obter as curvatures e direções principais que serão utilizadas como novas bases tangentes. A implementação do método de mínimos quadrados e iteração de Jacobi na GPU são triviais, uma vez que as operações envolvidas são operações aritméticas convencionais suportadas pela atual arquitetura de *hardware* gráfico.

Seguindo o paradigma de utilizar a GPU como um processador de fluxo de propósito geral, os dados da geometria e conectividade do modelo são armazenados previamente em texturas com formato de ponto flutuante. Essas texturas são acessadas no processador de fragmentos, que por sua vez armazena os resultados calculados em texturas utilizadas como alvo de renderização. O conteúdo dessas texturas é posteriormente utilizado pelas outras etapas da arquitetura de interação.

A seguir apresentamos os resultados dos testes de desempenho e robustez obtidos pelas nossas técnicas de estimativa de elementos de geometria diferencial discreta.

4.3.1 Elementos de primeira ordem

Para testar a abordagem proposta de estimativa de bases tangentes (vetor normal, vetor tangente e vetor bitangente), implementamos um conjunto de três *shaders* HLSL compatíveis com o modelo de *shader* 3.0. Cada *shader* realiza um passo do algoritmo descrito na seção 4.2.1. A listagem dos códigos-fonte é mostrada no apêndice A.

De modo a validar nossa técnica, implementamos uma aplicação de teste em C++/Direct3D que carrega na GPU uma geometria estática e, para cada quadro de exibição, desloca cada vértice ao longo de seu vetor normal. O valor de deslocamento é definido por uma textura volumétrica de ruído de Perlin amostrada no processador de vértices. O acesso à textura é feito de tal modo que um de seus eixos está associado ao tempo da animação. Assim, diferentes deslocamentos de geometria são produzidos para cada quadro de exibição. A geometria é renderizada com iluminação de Phong avaliada para cada *pixel*, e *normal mapping* com texturas definidas no espaço tangente. Para obter resultados satisfatórios de iluminação, utilizamos nossa técnica para recalcular as bases tangentes em cada quadro. Os modelos utilizados foram esferas com diferentes níveis de refinamento da malha. A aparência final de um desses modelos após a deformação e aplicação do *normal mapping* é mostrada na figura 4.1.

O desempenho de nosso algoritmo foi medido em um computador AMD Athlon 64 3500+ de 2.2GHz e 2GB RAM, equipado com uma placa gráfica NVIDIA GeForce 8800 GTX com 768MB. Para medir o tempo de processamento, não consideramos o tempo necessário a renderização com a finalidade de exibição dos objetos, uma vez que isto depende largamente das técnicas de iluminação e mapeamento de detalhes 3D utilizadas. Assim, nossas medições mostram apenas a sobrecarga do cálculo das bases tangentes.

O gráfico da figura 4.3 mostra uma comparação do tempo de processamento necessário para calcular, na GPU, bases tangentes completas (normal, tangente, bitangente) e apenas normais aos vértices, em um modelo com um número crescente de vértices. Como esperado, o tempo de processamento aumenta linearmente com relação ao número de vértices. O cálculo dos vetores tangente e bitangente adiciona uma sobrecarga de aproximadamente 40% ao cálculo isolado dos vetores normais aos vértices. Os tempos convergem para aproximadamente 0.2 milissegundos na geometria mais simples. Esta sobrecarga constante é associada ao ajuste dos estados de renderização para os três passos de renderização.

A figura 4.4 apresenta os tempos de processamento para calcular bases tangentes e vetores normais em um modelo de chaleira composto de 15.578 vértices e 30.800 faces, utilizando diferentes abordagens. Em (A) o cálculo é realizado inteiramente na CPU e os resultados são gravados na textura de atributos de vértices utilizada posteriormente na GPU; em (B) o cálculo é realizado na CPU mas os resultados não são transferidos para os mapas de atributos na memória de vídeo para serem

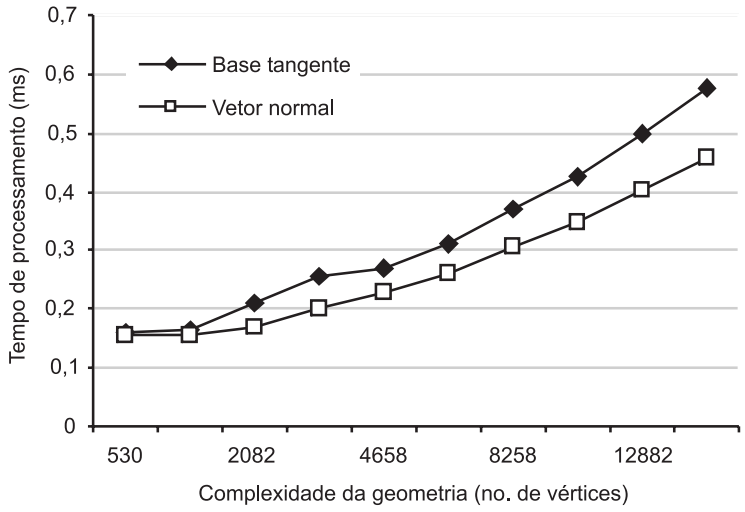


Fig. 4.3: Tempo de processamento para calcular bases tangentes na GPU para modelos com diferentes números de vértices.

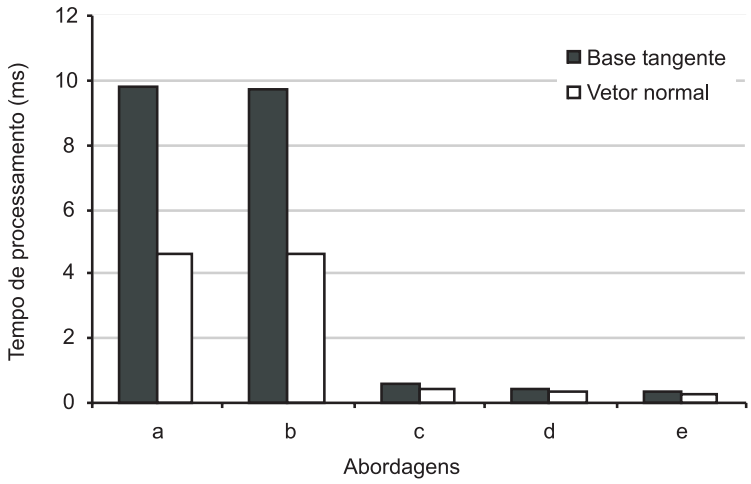


Fig. 4.4: Tempo de processamento do cálculo de bases tangentes segundo diferentes abordagens: (a) Na CPU; (b) Na CPU sem carregamento dos resultados na GPU; (c) Na GPU com valência máxima de vértices ilimitada; (d) Na GPU com valência máxima de vértices limitada a 8; (e) Na GPU com valência máxima de vértices limitada a 4.

utilizados em outras etapas da arquitetura de interação; em (C) o cálculo é realizado na GPU usando nossa técnica; em (D) o processamento é feito na GPU mas a valência máxima de vértices é limitada a 8, tornando possível compilar o *shader* sem o uso de instruções de fluxo de controle dinâmico; em (E) o cálculo também é realizado na GPU, com valência máxima de vértices limitada a 4. Os resultados da abordagem (B) são utilizados aqui apenas para mostrar o quanto a eficiência aumenta quando não está sendo considerada a sobrecarga resultante da transferência de dados entre a CPU e a GPU. De qualquer forma, tanto nos casos (A) e (B), o tempo de processamento é muito maior do que o tempo obtido do cálculo na GPU.

A eficiência em (D) e (E) aumenta porque o número de faces avaliadas é menor, mas também porque o laço utilizado para acumular os vetores tangentes às faces pode ser desenrolado completamente pelo compilador do *shader*. Em geral, malhas triangulares têm uma valência média de 6 vértices, porém com vértices isolados que podem conter até centenas de faces adjacentes [Kälberer et al., 2004]. A diferença de desempenho entre o algoritmo com valência arbitrária de vértices e valência de vértices limitada a 6 é de 25% (base tangente) a 30% (apenas vetor normal).

Em comparação com os mesmos algoritmos executados na CPU, a robustez das estimativas depende exclusivamente do formato numérico utilizado pelos registradores e texturas. Em nosso caso utilizamos o formato de ponto flutuante de 32 bits (IEEE 754, s23e8), conforme consta da especificação das APIs gráficas. Assim, os resultados são equivalentes aos produzidos por um algoritmo na CPU utilizando ponto flutuante de precisão simples.

4.3.2 Elementos de segunda e terceira ordem

Para validar nossa proposta de estimativa de elementos de geometria diferencial de segunda e terceira ordem considerando os requisitos apresentados na seção 4.2, realizamos testes de desempenho e testes de robustez sobre malhas triangulares obtidas de modelos digitalizados e malhas obtidas da amostragem de superfícies analíticas.

A implementação do algoritmo foi realizada tanto com processamento na CPU como processamento na GPU. Na CPU, o algoritmo foi implementado em C++ usando a biblioteca LAPACK de álgebra linear [Anderson et al., 1999]. Também implementamos com esta biblioteca os métodos de estimativa baseados apenas na vizinhança de 1-anel de vértices, como a aproximação por superfícies quadráticas, superfícies cúbicas [Goldfeather and Interrante, 2004], aproximação pela curvatura normal e média de tensores segundo Rusinkiewicz [2004]. Isto foi feito para uniformizar os métodos numéricos utilizados em comum pelos algoritmos citados e assim obter resultados mais confiáveis para os testes de robustez. Nesse caso, os métodos de resolução de sistema de equações lineares e extração de autovalores e autovetores é fornecido pela biblioteca LAPACK e é o mesmo para todos os algoritmos.

As figuras 4.5 e 4.6 exibem algumas imagens geradas por um aplicativo de visualização das propriedades de geometria diferencial de segunda e terceira ordem estimadas com o nosso algoritmo na GPU. Na figura 4.5, em (a) é mostrado o modelo de um nó com curvaturas principais visualizadas como cores. Em (b) é mostrada a curvatura Gaussiana $K = \kappa_{min}\kappa_{max}$ em tons de cinza sobre o modelo de uma chaleira. Nessa visualização, tons mais escuros e mais claros correspondem, respectivamente, a curvaturas negativas e positivas. O tom de cinza médio corresponde a curvatura zero. É possível observar que a superfície é localmente hiperbólica quando $K < 0$ (tons escuros), parabólica ou planar quando $K = 0$ (cinza médio), e elíptica quando $K > 0$ (tons claros). As direções principais dos modelos da chaleira e nó são mostradas em (c) e (d), respectivamente. As setas vermelhas apontam para as direções tangentes das curvaturas mínimas, enquanto que as setas verdes apontam para as direções tangentes das curvaturas máximas. Na figura 4.6, em (a) é visualizada a curvatura média $H = \frac{1}{2}(\kappa_{min} + \kappa_{max})$, em tons de cinza, sobre o modelo de um coelho. Em (b) é visualizada uma codificação em cores da magnitude do tensor de derivada de curvatura ($|\mathbb{C}_s| = c_1^2 + c_2^2 + c_3^2 + c_4^2$, onde c_1, c_2, c_3 e c_4 são os componentes não simétricos do tensor) para o modelo do coelho.

A figura 4.7 mostra o erro RMS da estimativa da curvatura Gaussiana em um toro, em termos do grau de irregularidade na amostragem. O toro, de raio interno 1 e raio externo 2, é discretizado a partir da superfície paramétrica:

$$S(u,v) = \begin{cases} x = (2 + \cos v)\cos u \\ y = (2 + \cos v)\sin u \\ z = \sin v \end{cases} \tag{4.25}$$

A discretização utilizada é composta de 30 lados na circunferência interna e 60 na circunferência externa, e utiliza normais aos vértices calculadas segundo o algoritmo de Gouraud [1971], que é o mesmo algoritmo utilizado em nossa estimativa de elementos de primeira ordem.

Para o cálculo do erro, utilizamos como parâmetro de comparação aquele obtido pela avaliação da equação analítica da curvatura Gaussiana nesse modelo:

$$K = \frac{\cos v}{2 + \cos v} \tag{4.26}$$

A figura 4.8 mostra duas renderizações do toro em *wireframe* para ilustrar a irregularidade dessa distribuição para os graus respectivos de 0% e 100%. O valor de 100% corresponde ao maior grau de aleatoriedade possível antes da introdução de triângulos degenerados. Todos os resultados foram obtidos com números em formato de ponto flutuante de precisão simples. Esse gráfico mostra que

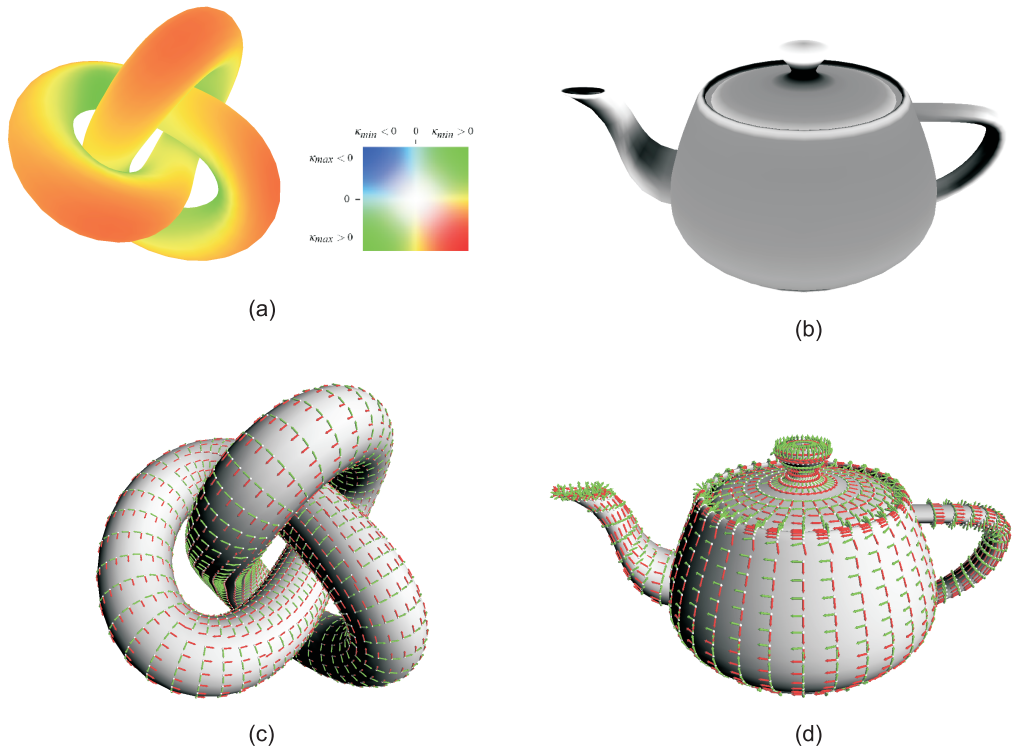


Fig. 4.5: Visualização de propriedades de geometria diferencial estimadas por nosso método na GPU. (a) Curvaturas principais exibidas como cores; (b) Curvatura Gaussiana, em tons de cinza; (c, d) Direções principais.

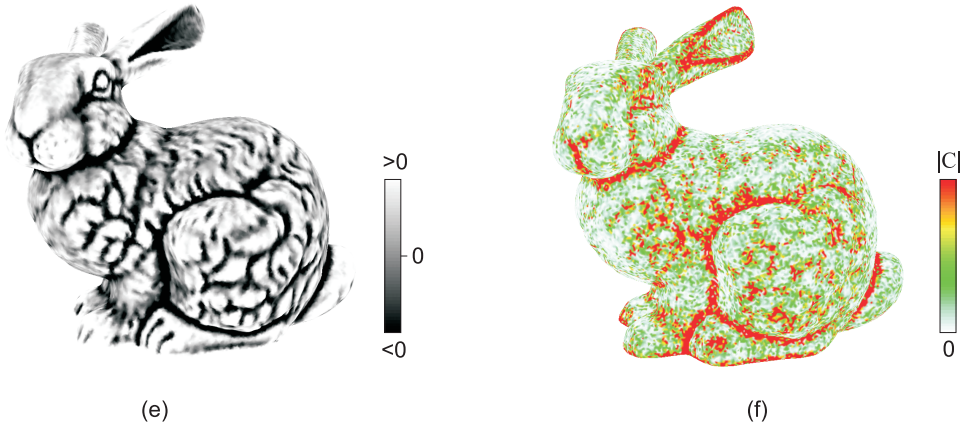


Fig. 4.6: Visualização de propriedades de geometria diferencial estimadas por nosso método na GPU. (a) Curvatura média, em tons de cinza; (b) Magnitude do tensor de derivada da curvatura.

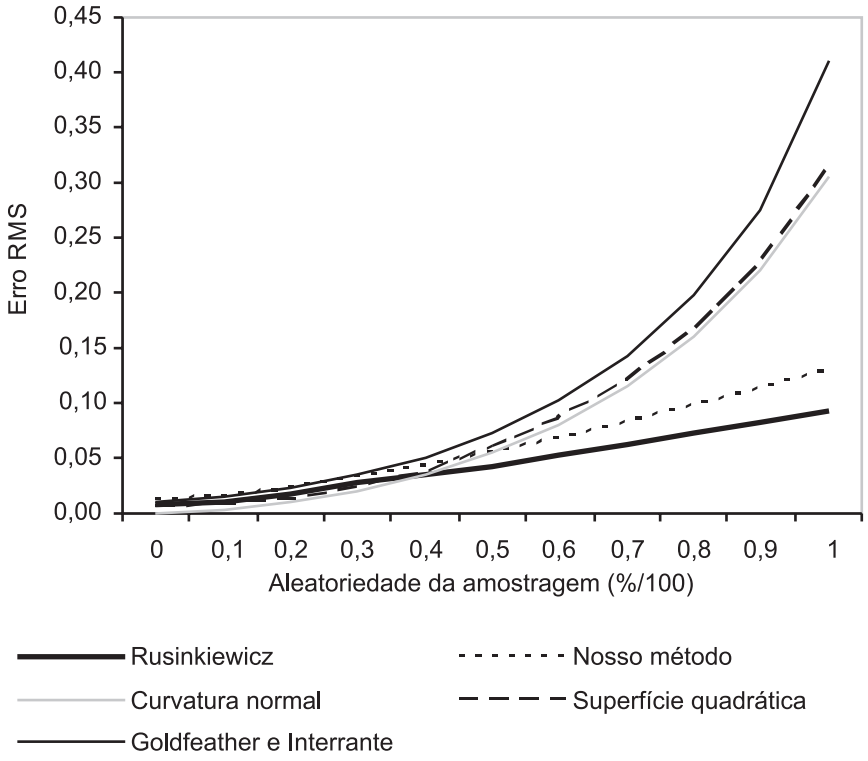


Fig. 4.7: Erro RMS da estimativa da curvatura Gaussiana em um toro discretizado em função de diferentes níveis de irregularidade da amostragem.

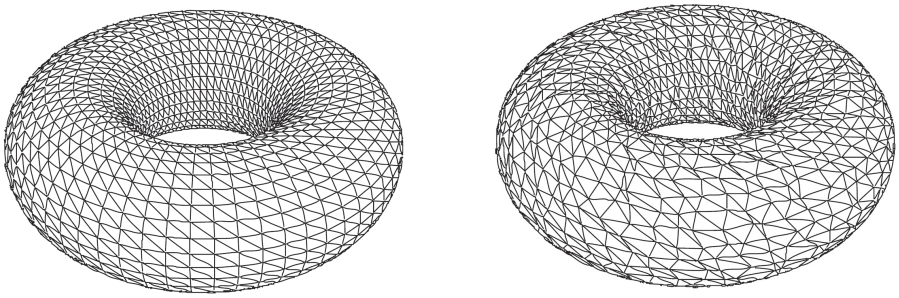


Fig. 4.8: Toro discretizado com regularidade máxima da malha (esquerda) e irregularidade máxima da malha antes da introdução de triângulos degenerados (direita).

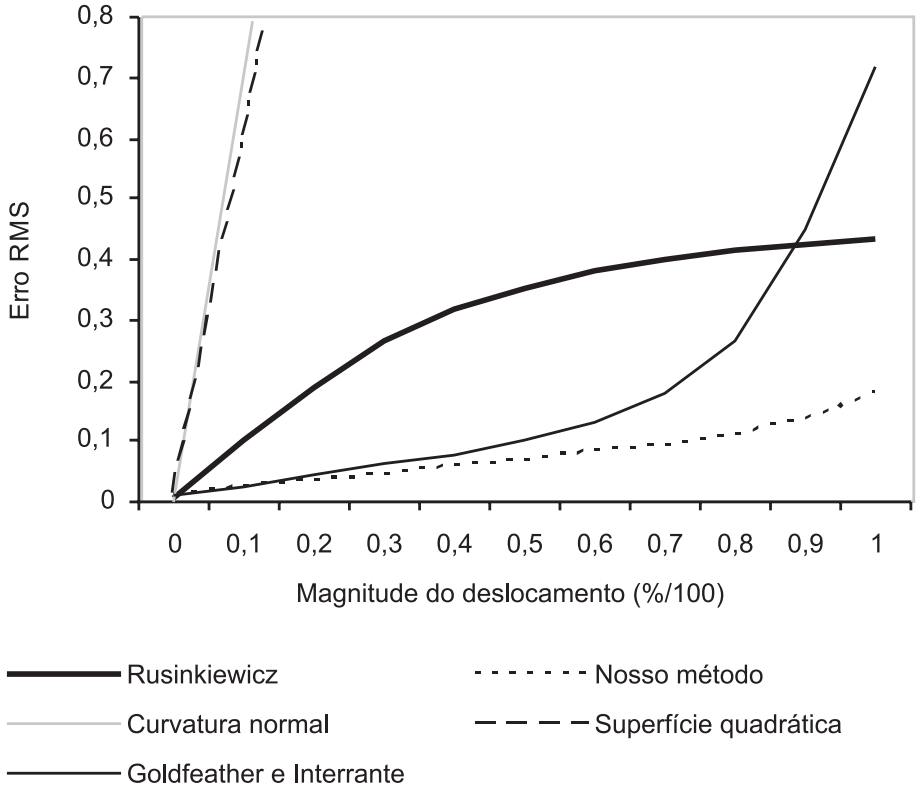


Fig. 4.9: Erro RMS da estimativa da curvatura Gaussiana em um toro discretizado segundo diferentes magnitudes de deslocamento dos vértices ao longo do vetor normal exato.

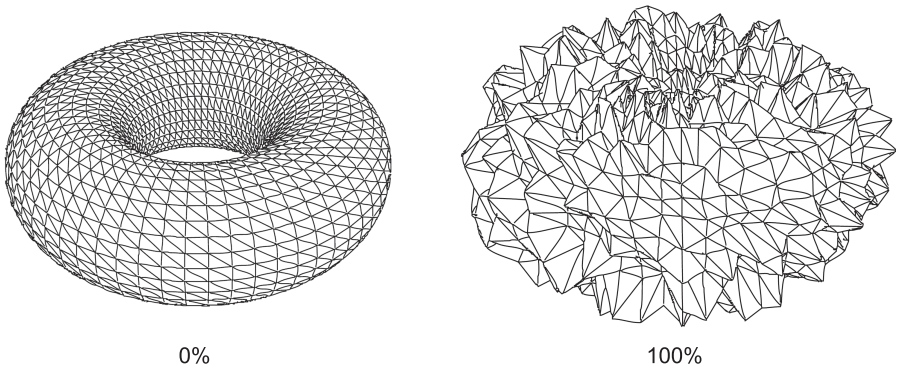


Fig. 4.10: Toro discretizado, com ruído de deslocamento aplicado para cada vértice. Esquerda: com magnitude mínima do deslocamento. Direita: com magnitude máxima do deslocamento.

o algoritmo de Rusinkiewicz [2004] tem o menor crescimento do erro de estimativa em proporção à irregularidade da malha. Nosso algoritmo produz resultados similares, embora ligeiramente menos robustos. É possível verificar que as técnicas de aproximação por superfícies analíticas (quadrática ou cúbica) e curvatura normal produzem resultados mais acurados para o toro amostrado regularmente. Isso ocorre porque tais técnicas assumem que a superfície amostrada é localmente uma superfície polinomial, condição esta satisfeita para o toro. Por outro lado, ainda assim essas técnicas são mais sensíveis à irregularidade da amostragem da malha.

A figura 4.9 compara o erro RMS da estimativa da curvatura Gaussiana do toro discretizado, porém com ruído de deslocamento adicionado aleatoriamente a cada vértice. Cada vértice é deslocado em diferentes níveis de magnitude ao longo do vetor normal calculado analiticamente. Uma visualização do modelo em *wireframe* com os deslocamentos de magnitude mínima e máxima é mostrado na figura 4.10.

Nossa técnica produz os resultados mais acurados para este caso, superando os resultados obtidos com o algoritmo de Rusinkiewicz [2004]. Em contraste, as técnicas de aproximação por superfície quadrática e aproximação pela curvatura normal produzem erros grosseiros de estimativa mesmo com uma baixa porcentagem de adição de ruído. Esse resultado é esperado, pois tais técnicas não utilizam os vetores normais para contrabalançar o erro do deslocamento. Por outro lado, a técnica de Goldfeather and Interrante [2004], também baseada na aproximação por superfície analítica, utiliza os vetores normais disponíveis e portanto produz erros menores.

Em modelos digitalizados (*e.g.*, obtidos através de um *scanner* 3D) a malha é geralmente irregular e os vértices possuem diferentes valências. Além disso, não há um modelo analítico com o qual seja possível realizar uma comparação de robustez. Para realizar uma avaliação da robustez da estimativa sobre esses modelos, visualizamos as curvaturas principais codificadas em cores de modo a perceber visualmente a presença de *outliers*, *i.e.*, estimativas pontuais que divirjam de forma substancial das estimativas vizinhas.

Para os testes com modelos digitalizados, utilizamos o modelo de um cavalo com 48.484 vértices e 96.964 triângulos, e de um coelho (*Stanford bunny*) com 72.027 vértices e 144.046 triângulos. Os modelos utilizados foram obtidos de Rusinkiewicz et al. [2007].

A figura 4.11 mostra uma visualização das curvaturas principais codificadas como cores para o modelo do cavalo segundo diferentes técnicas de estimativas das curvaturas principais. As cores são moduladas com o sombreamento difuso do modelo. A visualização correspondente para o modelo do coelho é mostrada na figura 4.12. A presença de *outliers*, visualizados como pequenos pontos coloridos e isolados, é relativamente baixa em nossa técnica e visualmente equivalente ao resultado obtido pela técnica de Rusinkiewicz [2004]. As técnicas de aproximação por superfícies analíticas e curvatura normal se revelam inferiores nesses modelos digitalizados porque tais modelos são geral-

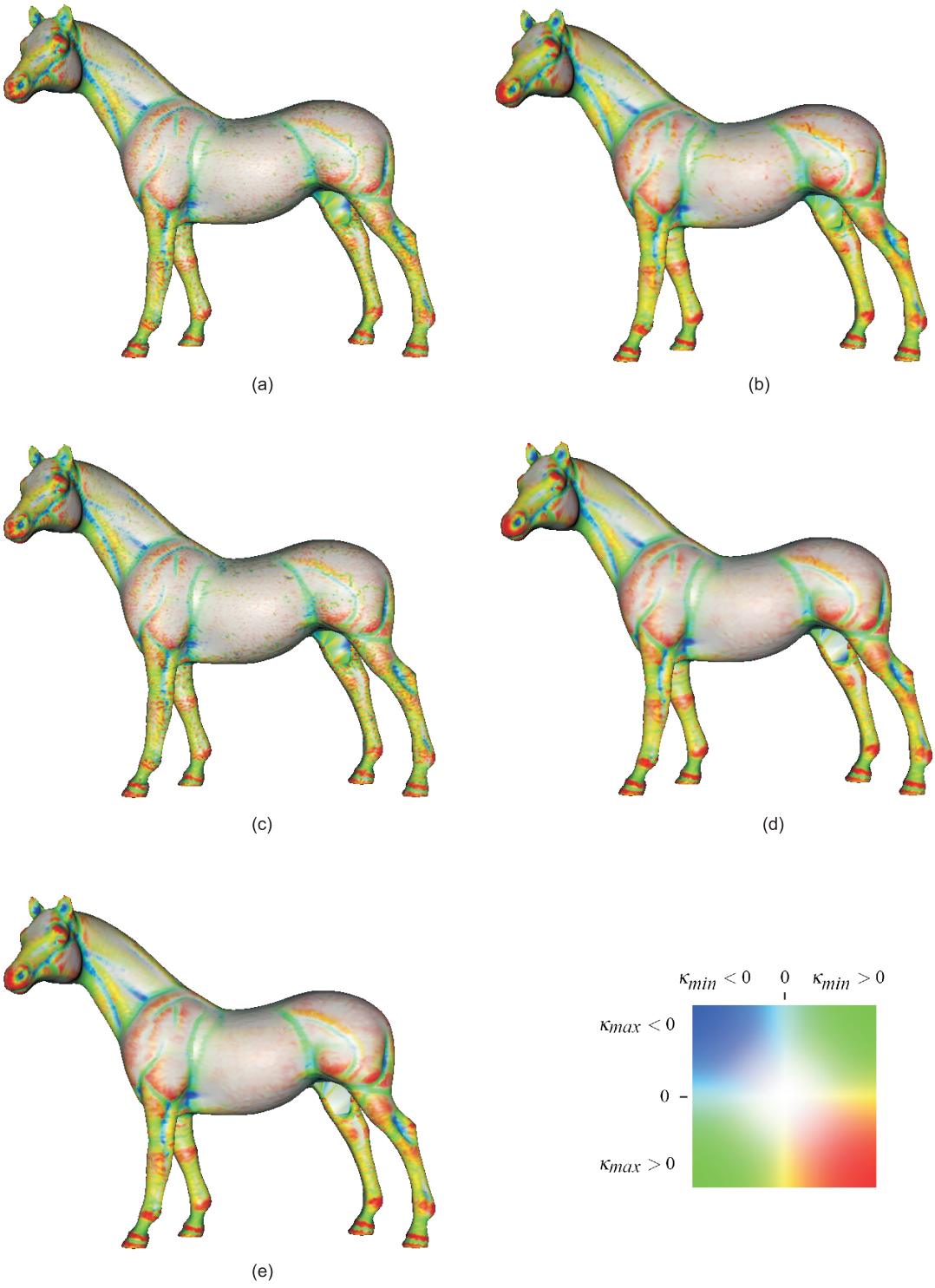


Fig. 4.11: Visualização das curvaturas principais codificadas como cores, para o modelo de um cavalo. (a) Aproximação por superfícies quadráticas; (b) Aproximação por superfícies cúbicas [Goldfeather and Interrante, 2004]; (c) Aproximação por curvatura normal; (d) Aproximação pela média do tensor de curvatura [Rusinkiewicz, 2004]; (e) Nossa técnica.

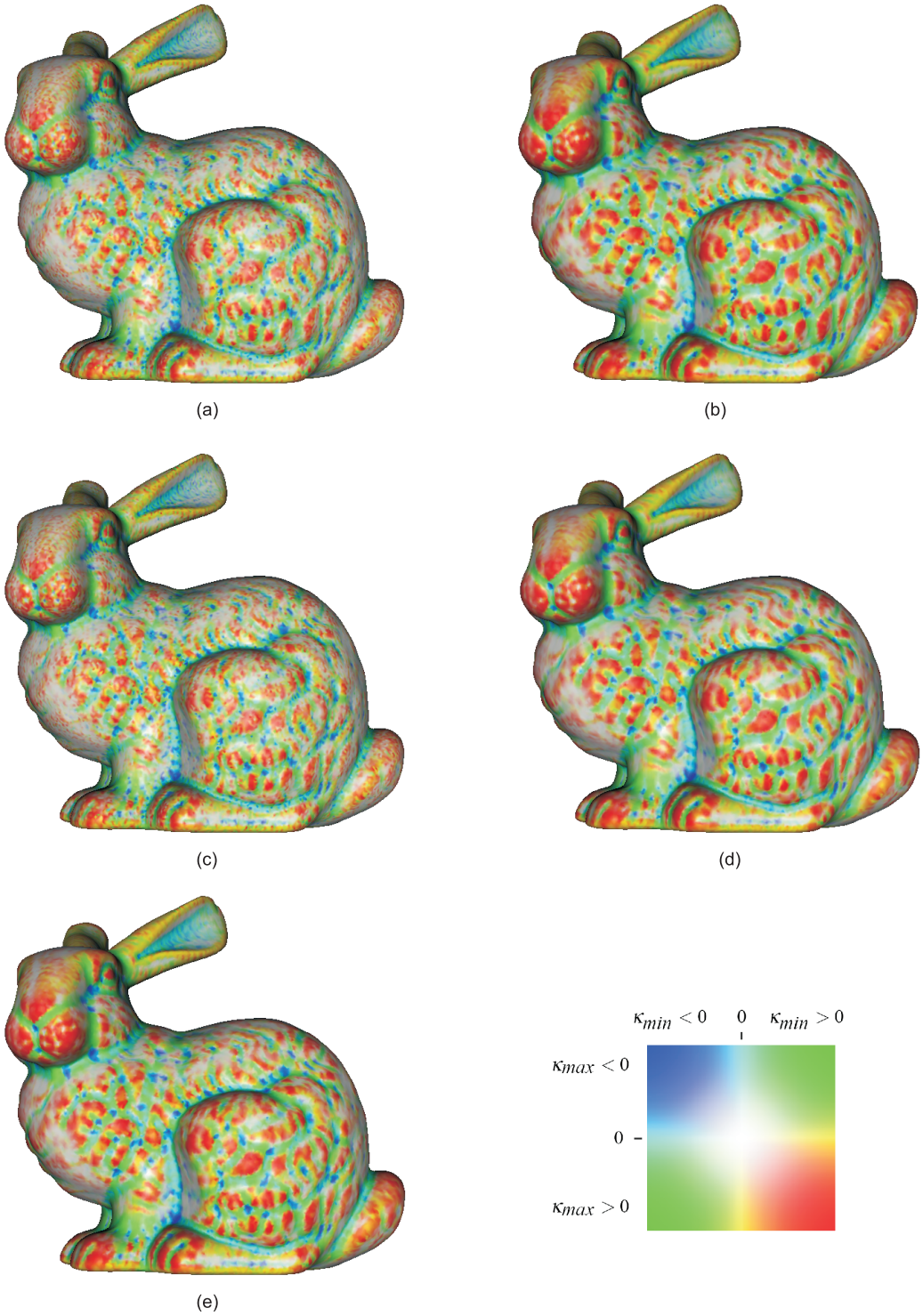


Fig. 4.12: Visualização das curvaturas principais codificadas como cores, para o modelo de um coelho. (a) Aproximação por superfícies quadráticas; (b) Aproximação por superfícies cúbicas [Goldfeather and Interrante, 2004]; (c) Aproximação por curvatura normal; (d) Aproximação pela média do tensor de curvatura [Rusinkiewicz, 2004]; (e) Nossa técnica.

mente representados por malhas irregulares e contém regiões locais que, conforme relatamos nas seções 4.1.2 e 4.1.2, são consideradas problemáticas para essas técnicas. As técnicas de aproximação por superfícies quadráticas e aproximação pela curvatura normal produzem resultados visualmente idênticos, o que confirma a suposição de que os dois métodos sofrem dos mesmos problemas de robustez.

Para realizar os testes de desempenho, comparamos nossa técnica com a técnica de Rusinkiewicz [2004] em razão da similaridade das formulações utilizadas e dos resultados de robustez obtidos. Neste caso, ambos os algoritmos foram implementados utilizando código C++ otimizado. Em particular, o algoritmo de Rusinkiewicz [2004] foi obtido da biblioteca *trimesh2* [Rusinkiewicz, 2006]. A plataforma dos testes de desempenho foi a mesma utilizada nos testes dos algoritmos de estimativa de elementos de primeira ordem.

A figura 4.13 mostra o desempenho obtido para calcular o tensor de curvatura (incluindo curvaturas e direções principais) e tensor de derivada de curvatura. O gráfico mostra os resultados obtidos com o algoritmo de Rusinkiewicz [2004], e nosso algoritmo implementado tanto na CPU como na GPU. O tempo de processamento de nosso método na GPU já inclui o tempo gasto para realizar a transferência do resultado da GPU para a CPU. Entretanto, o gráfico não exibe o tempo necessário para pré-processar as estruturas de dados utilizadas em cada técnica, uma vez que isso pode ser feito em uma etapa de pré-processamento se considerarmos que a topologia do modelo não sofrerá modificações na GPU. A medição desse tempo adicional é mostrada na figura 4.14. Em nosso método, este é o tempo necessário para gerar a lista de vizinhança de 1-anel de cada vértice e o tempo gasto para carregar esses dados à GPU. Para o método de Rusinkiewicz [2004], é o tempo necessário para pré-calcular as áreas das regiões de Voronoi em torno de cada vértice. Nesse caso, nosso método na GPU possui a maior sobrecarga de pré-processamento como resultado do carregamento das texturas para a GPU.

4.4 Considerações finais

Vimos no capítulo 3 que as propriedades geométricas diferenciais são suficientes para a implementação de tarefas de seleção e posicionamento restrito analisadas. De acordo com a arquitetura do atual *hardware* gráfico programável (figura 2.3), há dois estágios de processamento programável ao longo do fluxo de renderização: processamento de vértices e processamento de fragmentos. A nível do processador de vértices, o atributo essencial para visualização é o atributo de posição do vértice. Atributos adicionais podem ser designados pela aplicação, como vetor normal e coordenadas de textura. No processador de fragmentos, os atributos são os mesmos fornecidos ao processador de vértices, interpolados linearmente ao longo da primitiva formada pelos vértices. O algoritmo de

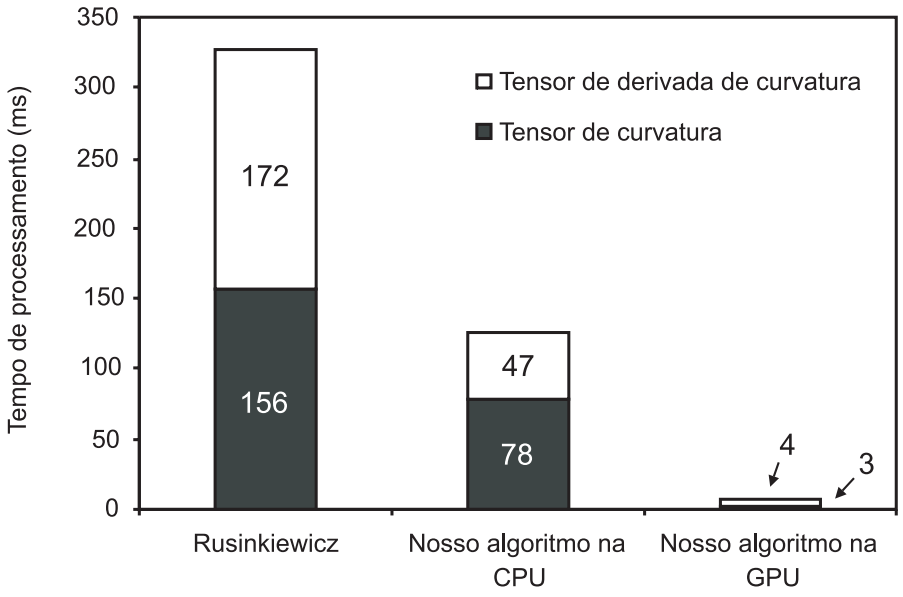


Fig. 4.13: Tempo de processamento, em milissegundos, para estimar o tensor de curvatura e tensor de derivada de curvatura para o modelo do cavalo.

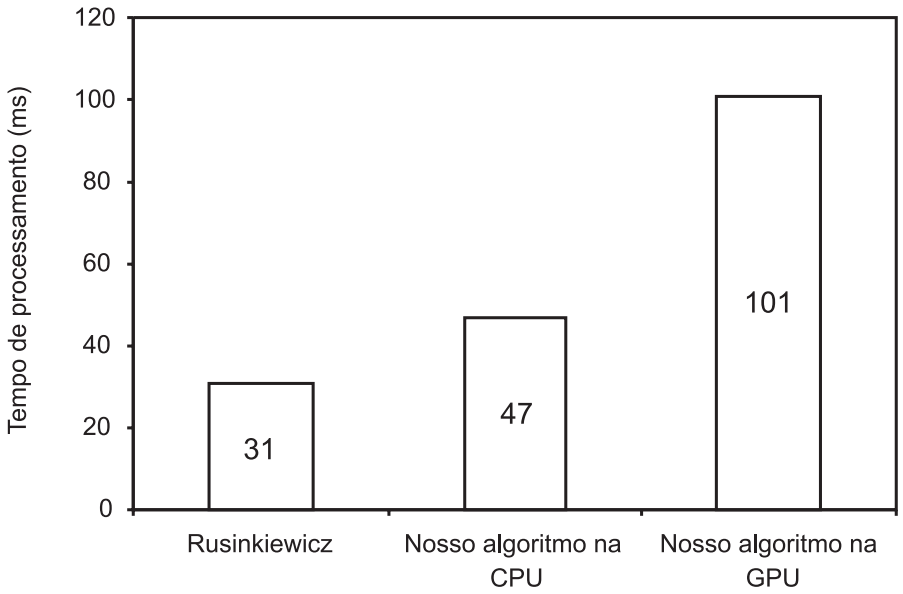


Fig. 4.14: Tempo de processamento, em milissegundos, para pré-processar as estruturas de dados utilizadas em cada método de estimativa.

estimativa apresentado neste capítulo requer como atributos de entrada a posição dos vértices, e coordenadas de textura para o cálculo dos vetores tangente e bitangente. Além destes atributos, requer também informações de conectividade da geometria, tais como a vizinhança de 1-anel de cada vértice. Embora tais informações não estejam disponíveis no processador de vértices como atributos dos vértices, podemos armazenar tais dados em texturas que podem ser amostradas nestes processadores em GPUs compatíveis com o modelo de *shader* 3.0. Dessa forma, podemos realizar a estimativa das propriedades de geometria diferencial de modo a levar em consideração as deformações geométricas ocorridas no processador de vértices. Entretanto, os algoritmos apresentados não são adequados para realizar a estimativa das propriedades de geometria diferencial após a rasterização, pois isto requer o conhecimento dos atributos na vizinhança de cada fragmento.

Dos resultados apresentados, verificamos que nosso algoritmo apresenta um ganho de desempenho importante em relação à técnica na qual nos baseamos. Isso se explica principalmente pela possibilidade de realizar a estimativa em apenas um laço. Aliada à capacidade de trabalhar em malhas arbitrárias, nossa proposta é ao mesmo tempo simples e adequada para a implementação nas atuais GPUs. De fato, conseguimos calcular tais propriedades em tempo real mesmo para modelos compostos de centenas de milhares de vértices. Esse aumento no desempenho possibilita que a técnica seja utilizada no contexto da arquitetura de suporte a tarefas de interação apresentada no capítulo 5. Embora os resultados aqui apresentados tenham sido obtidos através de uma placa gráfica NVIDIA GeForce 8800 GTX, testes utilizando uma placa gráfica NVIDIA GeForce 6200 e NVIDIA GeForce 7900 GTX também foram realizados. Os resultados destes testes estão disponíveis em Batagelo and Wu [2007a], e mostram que o comportamento do algoritmo é semelhante em todas essas arquiteturas, diferindo apenas por um valor constante.

Ainda que os testes tenham mostrado que a robustez de nossa técnica seja menor se comparada com a técnica de Rusinkiewicz [2004], acreditamos que essa redução é pouco significativa, conforme pudemos observar na baixa quantidade de *outliers* presentes na visualização da estimativa da curvatura em modelos digitalizados.

Capítulo 5

Arquitetura de interação

Neste capítulo apresentamos uma proposta de arquitetura de suporte à implementação de tarefas de manipulação direta 3D usando dispositivos apontadores 2D sobre geometria deformada na GPU.

A arquitetura é baseada em dois pressupostos principais:

1. O usuário espera interagir com aquilo que ele está vendo, ainda que o modelo exibido divirja de forma significativa do modelo original. Em outras palavras, assumimos que a manipulação direta de um modelo 3D deve atender à expectativa do usuário de interagir com o modelo após este ter sofrido as modificações realizadas no fluxo de visualização. Este pressuposto é a principal motivação para o desenvolvimento da arquitetura proposta, conforme exposto na seção 1.1.

2. A atual arquitetura de *hardware* gráfico programável é flexível o suficiente para processar os atributos geométricos necessários para a implementação de tarefas básicas de manipulação direta. Esse pressuposto é utilizado para desenvolvermos uma arquitetura de *software* de suporte eficiente à manipulação direta 3D de modelos geométricos modificados na GPU pelos processadores de vértices e fragmentos. Esta é a idéia básica e hipótese principal desta tese, sobre a qual defendemos a realização da arquitetura de interação 3D.

Acreditamos que uma interação *consistente* de manipulação direta 3D de modelos geométricos deformados na GPU é aquela que procura mapear de volta ao modelo original na CPU os atributos geométricos do modelo renderizado, como se o modelo original tivesse sido efetivamente deformado. Conforme já exposto, partimos do princípio de que o usuário espera interagir com aquilo que ele está vendo. Esse pressuposto relaciona o paradigma WYSIWYG (*What You See Is What You Get*) a duas das tarefas básicas de interação que são afetadas por geometria deformada na GPU: posicionamento e seleção.

WYSIWYG está relacionado ao modo como o usuário espera acessar o conteúdo de interesse da forma como ele está sendo apresentado visualmente. Particularmente para interação com modelos geométricos processados em *hardware* gráfico programável, é necessário levar em consideração a possível existência de modificações dos atributos dos vértices e fragmentos – modificações essas que não são propagadas ao modelo original submetido à GPU pela CPU, mas que podem alterar de forma significativa a aparência do modelo visualizado. Ainda assim, a arquitetura deve ser capaz de prover funcionalidades de interação com essa nova geometria através do cálculo dos atributos geométricos necessários para as tarefas básicas de manipulação direta 3D usando dispositivos apontadores 2D. Este caso pode requerer um desacoplamento completo do processamento da CPU à GPU para calcular atributos da geometria para interação, desacoplamento este limitado apenas pela dependência da CPU em tratar eventos do sistema de janelas e gerenciar a entrada e saída da GPU.

O restante deste capítulo está organizado da seguinte forma. Na seção 5.1 apresentamos os requisitos nos quais nos baseamos para propor a arquitetura. Na seção 5.2 situamos o escopo da arquitetura segundo uma classificação das camadas principais de abstração das componentes que formam uma interface gráfica. A idéia básica de armazenar atributos geométricos e não geométricos para cada *pixel* da imagem é detalhada na seção 5.3. O fluxo de processamento da arquitetura é mostrado na seção 5.4, juntamente com a especificação das interfaces de entrada e saída. O procedimento de uso dessa especificação, junto com um exemplo de aplicação, também é apresentado na seção 5.5.

5.1 Requisitos

Os requisitos nos quais nos baseamos para propor esta arquitetura são fundamentados na necessidade de fazer com que o desenvolvedor de interfaces gráficas tenha meios de implementar rapidamente tarefas de manipulação direta 3D sobre geometria processada pelas atuais GPUs, sem contudo impor restrições quanto ao uso do sistema de janelas ou sistema de gerenciamento de interfaces gráficas utilizado. Dessa forma, o uso da arquitetura é voltado ao desenvolvedor dessas aplicações de interfaces gráficas. Os requisitos para tornar isso possível são os seguintes:

- **Generalidade**. O requisito de generalidade é aplicado tanto aos tipos de modificações que os modelos podem sofrer no *hardware* gráfico, quanto aos tipos de primitivas com os quais a manipulação direta é realizada, e os atributos geométricos ou não geométricos que podem ser obtidos desses modelos.

  A arquitetura proposta deve ser capaz de trabalhar, de forma transparente ao desenvolvedor das tarefas de interação, com todas as primitivas suportados pela atual arquitetura de *hardware* gráfico. Além disso, deve considerar todas as possibilidades de deformação dessas primitivas

durante o processamento realizado no fluxo de renderização. Idealmente, devem ser consideradas deformações com relação aos vértices e com relação aos fragmentos produzidos durante a rasterização. Em razão da complexidade envolvida no tratamento de geometria que sofreu deformações com relação aos seus fragmentos, restringimos o escopo da arquitetura de modo a tratar, de forma transparente à aplicação, somente deformações arbitrárias dos vértices. Para a modificação de atributos de fragmentos, a arquitetura possibilita que a aplicação forneça seus próprios procedimentos de tratamento de tais modificações.

Diferentes tipos de atributos geométricos devem ser calculados na GPU para representações de superfícies suaves por malhas triangulares de modo a tornar possível a realização de um mapeamento entre os pontos da superfície renderizada na tela e os pontos da superfície original (*i.e.*, antes das deformações na GPU), ou ainda a reconstrução, na CPU, da geometria deformada a partir da obtenção de propriedades de geometria diferencial dos pontos da superfície amostrados durante a renderização. A arquitetura também deve permitir que a aplicação escolha essas propriedades a partir de uma paleta de atributos geométricos calculados na GPU, e possibilitar a adição de novos atributos com semântica definida pela aplicação.

- **Eficiência**. A integração da arquitetura de interação com a atual arquitetura de *hardware* gráfico deve herdar as vantagens provenientes do uso da GPU como um processador de fluxo de propósito geral: alto poder computacional e processamento assíncrono com relação à CPU. Para isso, o processamento associado à estimativa das propriedades de geometria diferencial deve ser realizado completamente na GPU usando a geometria armazenada na memória de vídeo local. Tal estimativa exige o acesso a informações de conectividade da malha triangular, informações essas que geralmente não são fornecidas automaticamente pelas atuais GPUs (GPUs compatíveis com o modelo de *shader* 4.0 suportam tal acesso, mas ainda de forma limitada). Para contornar essa limitação, tanto os atributos geométricos como atributos topológicos devem ser armazenados na memória de vídeo, organizadas como texturas em formato de ponto flutuante. Neste caso, as texturas são utilizadas como áreas de memória genérica, e contêm dados como a posição dos vértices, vetor normal e índices de vértices adjacentes.

  A obtenção dos resultados do processamento de atributos na GPU requer transferências de dados entre a GPU e a CPU através do barramento AGP (*Advanced Graphics Port*) ou PCI (*Peripheral Component Interconnect*) *Express*. Essa transferência pode ser ineficiente caso uma grande quantidade de dados precise ser transferida a cada quadro de exibição. Além disso, requer a sincronização de processamento entre as duas unidades de processamento, o que reduz o desempenho associado ao processamento paralelo. Supomos que, em nossa arquitetura, essa característica não representa um problema significativo, porquanto a quantidade de *pixels* lidos

do *buffer* de renderização é geralmente desprezível (atributos de apenas um *pixel* para a maioria das tarefas de interação), e o impacto da sincronização pode ser reduzido, embora não eliminado, através da diminuição de freqüência de leitura dos atributos do *frame buffer*, sem prejuízo à sensação de continuidade da interação. Segundo Olsen [1998], a taxa mínima de atualização nesses casos pode ser de apenas 5 FPS (*Frames Per Second* - Quadros Por Segundo), ainda que a taxa de visualização mínima esteja situada entre 20 e 30 FPS.

- **Robustez**. A arquitetura deve empregar técnicas acuradas de estimativa das propriedades de geometria diferencial de representações de superfícies suaves por malhas triangulares, uma vez que esses resultados serão utilizados para realizar as tarefas de seleção e posicionamento com restrições.

  Uma vez que consideramos o uso da GPU para realizar as estimativas das propriedades geométricas, tais atributos poderão ser processados com uma precisão numérica correspondente às diferentes opções de precisão dos registradores (*e.g.*, 16 ou 32 bits), e segundo os formatos dos *buffers* de renderização suportados. Atualmente, a precisão máxima obtida em tais *buffers* é de 32 bits por componente de cor, de modo que, mesmo que o processamento seja feito em 64 bits, o atributo final tem a precisão reduzida a 32 bits. As atuais especificações dos modelos de *shaders* também consideram a possibilidade do uso de valores de ponto flutuante de 16 ou 64 bits. Entretanto, o suporte nativo a ponto flutuante com tais precisões não é encontrado em todas as GPUs. Neste caso, porém, é possível simular o comportamento usando ponto flutuante de 32 bits [Microsoft, 2006]. Infelizmente, na prática ainda não existe um consenso a esse respeito e algumas arquiteturas podem realizar todo o processamento com precisão até mesmo de 24 bits.

  Uma vez que consideramos que o usuário está interagindo com aquilo que ele está vendo, a precisão espacial pode ser determinada pela resolução da própria imagem visualizada, *i.e.*, pelo espaço discreto de *pixels* na tela. Desse modo, a precisão espacial está diretamente associada à resolução e à qualidade da imagem visualizada pelo usuário, o que é razoável segundo o pressuposto de que o usuário espera interagir com aquilo que ele vê.

- **Reusabilidade**. O critério de reusabilidade está relacionado à facilidade de uso e adaptação da arquitetura, de modo que diferentes tarefas de interação de manipulação direta 3D podem ser implementadas sem a necessidade de reimplementação de algoritmos de *ray picking* específicos para os diferentes modelos tratados. Refere-se ainda à característica de minimizar a necessidade de duplicação de código de modificação da geometria na CPU e GPU, seja para funções de deformação de vértices como para funções de deformação de fragmentos. Indiretamente, o preenchimento do critério de generalidade está relacionado ao critério de reusabilidade: uma

arquitetura que se adapta facilmente a diferentes tipos de funções de deformação, primitivas e cálculos de atributos, permite sua rápida reutilização em novas aplicações e em sistemas de interação já existentes.

A reusabilidade também deve ser ampliada pela tentativa de minimizar a dependência da arquitetura de interação com relação ao sistema de janelas. Para realizar isso, o tratamento de eventos de entrada disparados pelo sistema de janelas ou sistema gerenciador de interface gráfica não é processado diretamente pela arquitetura proposta. Ao contrário, tais eventos são tratados pela própria aplicação que, por sua vez, pode utilizá-los para mudar as configurações da arquitetura de interação. Por exemplo, através do tratamento dos eventos de movimentação do cursor do dispositivo apontador, consideramos que seja possível definir uma *região de interesse* (em coordenadas da janela) sobre a qual se deseja que os atributos geométricos sejam calculados na GPU. Cabe à aplicação a tarefa de utilizar esses atributos para implementar a tarefa de interação. Dessa forma, a arquitetura depende apenas da arquitetura programável das GPUs e, portanto, das APIs utilizadas para comunicação com esse tipo de *hardware* gráfico.

5.2 Escopo

De acordo com a classificação de Myers [1995], são quatro as camadas principais de abstração sobre as componentes que formam uma interface gráfica: sistema de janelas, *toolkits*, ferramentas de alto nível e aplicação. A camada de *toolkits* é composta de sistemas que encapsulam *widgets* tais como menus, botões e barras de rolagem. Atualmente, é comum a existência de *widgets* específicos de área de desenho, utilizados pela aplicação para exibir or gráficos processados pela biblioteca GL. Por exemplo, o *toolkit* Xt/Motif fornece os *widgets* `GLwDrawingArea` (baseado no Xt) e `GLwMDrawingArea` (baseado no Motif) específicos para exibir gráficos renderizados com o OpenGL [Flanagan, 1994]. Do mesmo modo, o *toolkit* Qt fornece o *widget* `QGLWidget` [Dalheimer, 2002], enquanto o GTK+ fornece os *widgets* `GtkGLArea` para a versão em C e `GtkGLArea-` para a versão em C++ do GTK [Logan, 2001]. Os gráficos são exibidos nesses *widgets* de acordo com as configurações da janela e matrizes de transformação geométrica especificadas no OpenGL.

Ao contrário do que ocorre com *widgets* como botões, menus, barras de rolagem e caixas de texto, o tratamento de eventos sobre o *widget* de área de desenho vinculada à API gráfica não possui necessariamente uma semântica pré-definida pelo *toolkit*. Por exemplo, no *widget* correspondente à barra de rolagem, o *toolkit* pode estabelecer que o comportamento padrão de rolar uma barra horizontal é obtido através de uma seqüência de eventos produzidos a partir de ações de um *mouse*, tal como: (1) pressionar o botão esquerdo do *mouse* sobre a barra de rolagem; (2) deslocar o *mouse* à esquerda ou à direita; (3) liberar o botão pressionado. Semelhante comportamento pré-determinado

se aplica aos eventos de outros *widgets*. Para o *widget* de área de desenho, cabe à aplicação a tarefa de especificar quais eventos deverão ser considerados, e como esses eventos serão tratados para modificar o conteúdo da área de desenho. Em particular, as tarefas básicas de posicionamento e seleção em manipulação direta 3D podem ser realizadas através do tratamento de eventos de movimentação do dispositivo apontador 2D e eventos de pressionamento de seus botões. A arquitetura proposta auxilia o desenvolvedor das ferramentas de alto nível ou desenvolvedor da aplicação a implementar rapidamente tais tarefas de interação utilizando apenas os atributos geométricos e não geométricos disponíveis para cada *pixel* do modelo renderizado, conforme sugerido pelo estudo de casos apresentado no capítulo 3.

Nossa arquitetura reproduz o processamento análogo à realização de um teste de interseção entre raios de seleção que partem do observador, atravessam os *pixels* da tela contidos numa região de interesse definida pela aplicação, e são propagados pelo volume da cena contido no volume de visualização. Os atributos calculados são essencialmente os mesmos atributos geométricos ou não geométricos que poderiam ser obtidos de uma superfície no ponto de interseção mais próximo entre um raio de seleção e a geometria usando um algoritmo de *ray picking* tradicional na CPU. Embora a arquitetura forneça dados apenas do ponto de interseção visível, *i.e.*, do ponto mais próximo do plano de projeção, é possível a obtenção de atributos em todas as interseções ao longo de cada raio de seleção, classificados segundo a distância dos pontos ao plano de projeção. Isto é obtido através da execução de múltiplos passos de renderização dos atributos da cena, de modo que, em cada passo de renderização adicional, são filtrados os fragmentos das primitivas visíveis dentro da região de interesse.

Os atributos geométricos e não geométricos processados pela arquitetura de interação são escritos em *buffers* de renderização não visíveis e codificados como componentes de cor dos *pixels* da imagem renderizada. Posteriormente, esses atributos são decodificados pela CPU e retornados à aplicação. Cabe à aplicação a tarefa de utilizar tais atributos de acordo com o tipo de tarefa de interação que está sendo implementado. Isso significa que, embora integrada ao laço de visualização da aplicação, a arquitetura não é responsável pela etapa de retorno visual da interação. Portanto, ela não é responsável pela execução de tarefas tais como atualizar a cena original ou modificar a posição e orientação de um *widget* 3D. Ao contrário, os atributos necessários para fazer essas modificações devem ser requisitados e utilizados pela aplicação de modo a fornecer um retorno visual particular à tarefa que está sendo implementada (*e.g.*, atualizando a posição e orientação do cursor 3D, mudando a aparência do modelo envolvido na interação, entre outras formas de realimentação). Desse modo, se a aplicação implementa uma ação de manipulação que modifica a geometria ou topologia do modelo, a própria aplicação é responsável pela atualização dos novos dados para a arquitetura usando uma interface específica de dados de entrada.

A arquitetura utiliza quaisquer primitivas suportadas pelo *hardware* gráfico atual. Assim, as possíveis representações de superfícies incluem malhas triangulares, primitivas baseadas em pontos e *voxels*, desde que tais representações sejam construídas com base nas primitivas suportadas em *hardware*. Por outro lado, o cálculo de propriedades de geometria diferencial discreta é realizado apenas sobre vértices, sejam eles de primitivas baseadas em pontos ou malhas triangulares. Para que a arquitetura de interação possa extrair relações de adjacência necessárias para o cálculo de propriedades de geometria diferencial, a geometria enviada à arquitetura deve ser apresentada na forma de um *buffer* de vértices e um *buffer* de índices de modo a formar uma malha triangular ou nuvem de pontos.

Técnicas de mapeamento de detalhes 3D são suportadas desde que sejam construídas sobre modelos criados com as primitivas citadas. Uma vez que a estimativa das propriedades de geometria diferencial é realizada apenas com relação aos vértices, a técnica de mapeamento de detalhes torna-se responsável pela atualização desses atributos caso eles sejam modificados durante a aplicação do detalhe.

A definição de atributos relativos às faces ou arestas só pode ser feita através da atribuição de atributos dos vértices que compõem essas primitivas. Por exemplo, identificadores de faces devem ser adicionados como atributos dos vértices que compõem cada face. Entretanto, a identificação de faces e arestas não pode ser feita com geometria indexada na qual um único vértice é compartilhado por mais de uma face ou aresta. Em tais casos, a aplicação deve fornecer uma geometria na qual os vértices coincidentes sejam repetidos. Assim, embora coincidentes na posição, seus atributos de identificação de face podem ser diferentes. A arquitetura considera esse tipo de geometria como geometria não indexada.

5.3 Codificação de atributos no domínio da imagem

A idéia básica da arquitetura proposta consiste em utilizar o fluxo de visualização já existente da atual arquitetura de *hardware* gráfico programável (figura 2.3) para processar atributos geométricos e não geométricos dos modelos renderizados. Tais atributos podem ser calculados para todos os *pixels* do modelo renderizado e armazenados no espaço discreto da tela em *buffers* de renderização não visíveis. Assim, cada *pixel* de um *buffer* de renderização não visível pode conter um valor que indica, por exemplo, o vetor normal da superfície naquele ponto, ou o identificador do objeto geométrico que contém o *pixel*. A arquitetura disponibiliza esses atributos à CPU para que eles sejam utilizados posteriormente na aplicação. Isto possibilita a implementação de tarefas eficientes de manipulação direta 3D que satisfaçam a consistência com o pressuposto de que o usuário espera interagir com o que ele está efetivamente visualizando.

A integração da arquitetura de interação com a arquitetura de *hardware* gráfico permite que os mesmos *shaders* de deformação utilizados na visualização podem ser utilizados também para levar em consideração as modificações sofridas ao longo do fluxo de visualização pelos atributos geométricos necessários em manipulação direta 3D. Essa estratégia evita ainda a duplicação de dados de geometria na CPU, uma vez que o mesmo modelo usado para visualização – armazenado em memória de vídeo local – também pode ser utilizado para o processamento de interação. Além disso, em razão do alto poder computacional e capacidade de processamento paralelo das atuais GPUs, é possível obter um aumento no desempenho do cálculo desses atributos mesmo em cenas que não sofrem deformações no fluxo de visualização. Em resumo, a arquitetura proposta pode ter desempenho superior ao obtido com o algoritmo tradicional de *ray picking*, mas ao mesmo tempo fornece uma solução satisfatória para o problema de interação com geometria deformada na GPU.

Os atributos processados pela GPU são codificados como componentes de cor em *buffers* de renderização não visíveis com formato em ponto flutuante, e decodificados pela CPU de modo a fornecer à aplicação os atributos correspondentes de cada *pixel* do modelo renderizado. Os seguintes atributos podem ser calculados para cada vértice e interpoladas para cada fragmento:

- **Valor de profundidade**. Profundidade do fragmento da superfície renderizada, em coordenadas relativas à janela. O cálculo desse atributo exige a existência, na geometria original, do atributo de posição 3D do vértice em coordenadas locais (coordenadas relativas ao espaço do objeto). Para transformações perspectivas obtidas em OpenGL com o comando `glFrustum()` ou `gluPerspective()` (da biblioteca GLU - *OpenGL Utility Library*), esse valor de profundidade é calculado como

$$z_{window} = \left[\frac{Z_f + Z_n}{Z_f - Z_n} + \frac{2Z_f Z_n}{z_{eye}(Z_f - Z_n)}\right]\left[\frac{Z_{f_{vp}} - Z_{n_{vp}}}{2}\right] + \frac{Z_{f_{vp}} + Z_{n_{vp}}}{2}, \tag{5.1}$$

  onde z_{eye} é o valor de profundidade relativo ao espaço da câmera, e Z_n e Z_f correspondem, respectivamente, aos valores de profundidade do plano de recorte próximo (*near clipping plane*) e distante (*far clipping plane*), também em coordenadas relativas ao espaço da câmera. As variáveis $Z_{n_{vp}}$ e $Z_{f_{vp}}$ correspondem aos valores utilizados pela API gráfica para mapear linearmente as coordenadas de profundidade já recortadas e divididas pela coordenada homogênea (normalizadas entre -1 e 1 no OpenGL, e entre 0 e 1 no Direct3D), para valores de profundidade do espaço da janela normalizados geralmente entre $Z_{n_{vp}} = 0$ e $Z_{f_{vp}} = 1$, onde $Z_{n_{vp}}$ corresponde ao valor do plano de recorte próximo e $Z_{f_{vp}}$ corresponde ao valor do plano de recorte distante. Em OpenGL, tais valores são definidos pelo comando `glDepthRange()`. Em Direct3D 9.0, são definidos através do ajuste das variáveis-membro `MinZ` e `MaxZ` da estrutura `D3DVIEWPORT9`, passada como parâmetro do método `IDirect3DDevice9::SetViewport`. Para transfor-

mações perspectivas obtidas em Direct3D com o comando `D3DXMatrixPerspectiveLH()` da biblioteca D3DX, z_{window} é calculado pela equação

$$z_{window} = \left[\frac{Z_f}{Z_f - Z_n} - \frac{Z_f Z_n}{Z_{eye}(Z_f - Z_n)}\right]\left[Z_{f_{vp}} - Z_{n_{vp}}\right] + Z_{n_{vp}}. \tag{5.2}$$

Por sua vez, em transformações ortográficas obtidas em OpenGL com o comando `glOrtho()`, ou com o comando `gluOrtho2D()` da biblioteca GLU, z_{window} é dado pela expressão

$$z_{window} = \frac{2z_{eye} + Z_f + Z_n}{Z_f - Z_n}\left[\frac{Z_{f_{vp}} - Z_{n_{vp}}}{2}\right] + \frac{Z_{f_{vp}} + Z_{n_{vp}}}{2}. \tag{5.3}$$

Como regra geral, tanto para transformações ortográficas como perspectivas em OpenGL e Direct3D, o primeiro termo entre colchetes nas equações 5.1 e 5.2 é substituído pelo termo obtido da transformação de z_{eye} pela matriz de projeção utilizada em cada API. Os demais termos (com os valores $Z_{n_{vp}}$ e $Z_{f_{vp}}$) são responsáveis apenas pelo mapeamento linear dos valores de profundidade segundo o alcance de profundidade definido na API.

Na CPU, este valor de profundidade pode ser utilizado pela aplicação para determinar a posição 3D do fragmento de superfície em coordenadas relativas ao espaço do objeto. Para isso, obtém-se a posição 2D do *pixel*, o valor de profundidade calculado e a componente de coordenada homogênea, transformando a coordenada homogênea resultante de volta ao espaço do objeto de acordo com a transformação inversa das matrizes da janela, projeção, visão e mundo. Em OpenGL, essa transformação é dada por

$$\begin{pmatrix} x_{object} \\ y_{object} \\ z_{object} \\ w_{object} \end{pmatrix} = M^{-1}P^{-1}V^{-1}\begin{pmatrix} x_{window} \\ y_{window} \\ z_{window} \\ w_{window} \end{pmatrix} = (PM)^{-1}V^{-1}\begin{pmatrix} x_{window} \\ y_{window} \\ z_{window} \\ w_{window} \end{pmatrix}, \tag{5.4}$$

onde V é a matriz da janela de visualização (*viewport*), P é a matriz de projeção (*projection*) e M é a matriz concatenada de mundo e visão (*modelview*). O procedimento equivalente se aplica às matrizes de transformação da API Direct3D. Para transformações perspectivas no OpenGL, o valor w_{window} corresponde ao valor negativo da coordenada z do espaço da câmera ($-z_{eye}$), e assim pode ser obtido de z_{window} isolando z_{eye} da equação 5.1, como se segue:

$$z_{eye} = \frac{Z_f Z_n (Z_{f_{vp}} - Z_{n_{vp}})}{Z_f - Z_n}\left[z_{window} - \frac{(Z_f + Z_n)(Z_{f_{vp}} - Z_{n_{vp}})}{2(Z_f - Z_n)} - \frac{Z_{f_{vp}} + Z_{n_{vp}}}{2}\right]^{-1}. \tag{5.5}$$

Este procedimento é realizado na CPU pelo comando `gluUnProject()` da biblioteca GLU

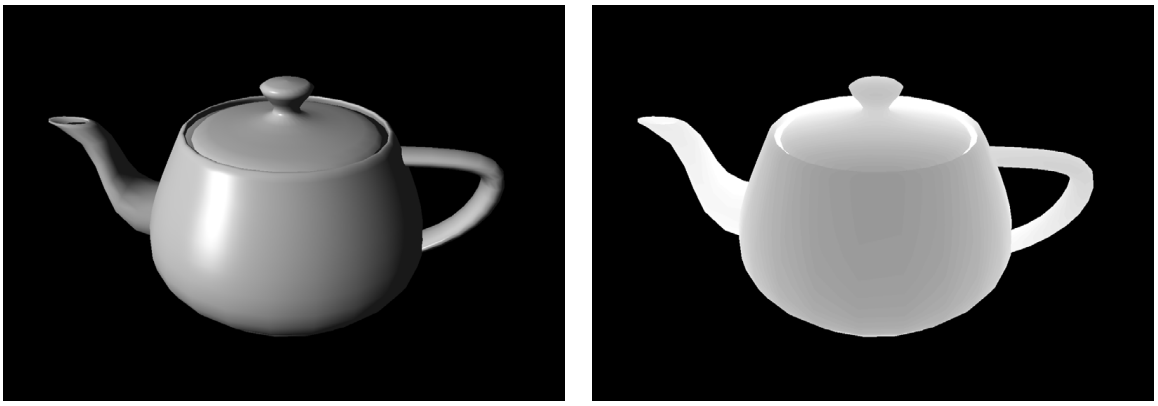


Fig. 5.1: Esquerda: objeto visualizado com modelo de iluminação de Blinn. Direita: visualização do valor de profundidade de cada *pixel*, mapeado em tons de cinza.

do OpenGL. Na API Direct3D, é utilizado o comando `D3DXVec3Unproject()`, da biblioteca D3DX. Neste caso, o valor w_{window} corresponde a z_{eye}. Para transformações ortográficas, $w_{window} = 1$.

Uma visualização do atributo de profundidade calculado para cada *pixel* no *buffer* de renderização não visível, e codificado em tons de cinza, é mostrado na figura 5.1. A intensidade do tom de cinza é proporcional à profundidade dos pontos da superfície: quanto mais claro, maior a profundidade. Como referência, o modelo renderizado com iluminação de Blinn [1977] também é mostrado.

- **Vetor normal**. O vetor normal pode ser fornecido pela aplicação em conjunto com os atributos do modelo original, ou pode ser calculado na GPU segundo uma adaptação do algoritmo tradicional de Gouraud [1971] para a arquitetura de processamento de fluxo.

  Cada coordenada é armazenada em uma componente de cor do *pixel* (*e.g.*, $\langle r, g, b\rangle = \langle x, y, z\rangle$ para um *buffer* de renderização com formato RGB). O vetor normal armazenado é sempre normalizado.

  Uma visualização desse atributo é mostrada na figura 5.2 em cores falsas. Nesta figura, as componentes do vetor são normalizadas e mapeadas para valores RGB através da equação $\langle r, g, b\rangle = (\langle x, y, z\rangle + \langle 1, 1, 1\rangle)/2$.

- **Vetores tangentes para mapeamento de detalhes 3D**. Tais vetores (vetor tangente e vetor bitangente) são perpendiculares entre si e alinhados segundo o mapeamento das coordenadas de textura no modelo. Juntamente com o vetor normal, formam uma base de coordenadas utilizada para mapear as coordenadas do espaço do objeto para as coordenadas do espaço de

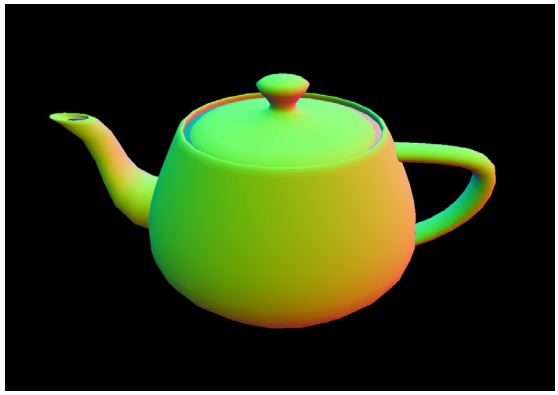


Fig. 5.2: Visualização do vetor normal em cores falsas.

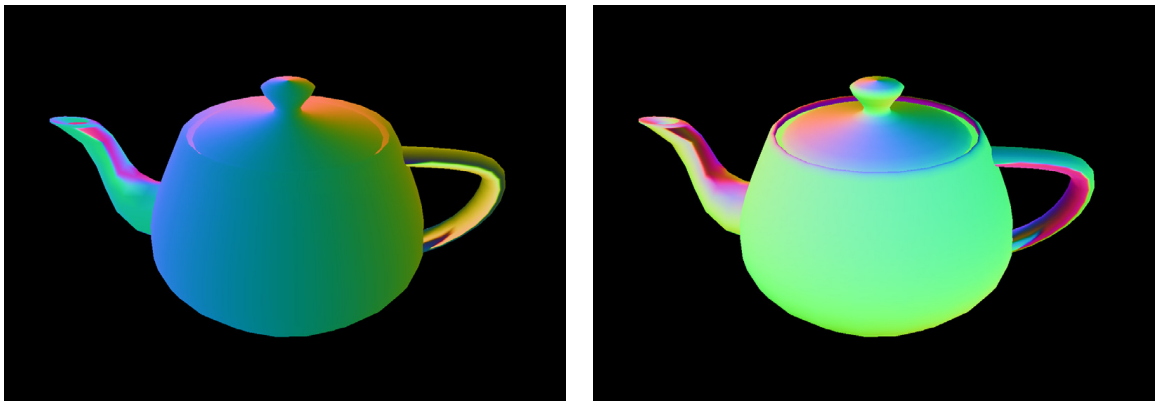


Fig. 5.3: Visualização dos vetores tangente (esquerda) e bitangente (direita) em cores falsas.

textura relativo à textura utilizada nas técnicas de mapeamento de detalhes 3D baseadas no uso de um espaço tangente. Cada componente de cada vetor é armazenada numa componente de cor, de forma semelhante à codificação do vetor normal.

A figura 5.3 mostra duas imagens que ilustram, em cores falsas, os valores dos vetores tangente e bitangente para cada *pixel*, usando o mesmo mapeamento para valores RGB utilizado na figura 5.2.

- **Coeficientes do tensor de curvatura**. Cada coeficiente e, f, g do tensor $\mathbb{II}_s$ é armazenado numa componente de cor do *buffer* de renderização não visível.

  A figura 5.4 mostra uma visualização do valor absoluto da soma dos coeficientes do tensor de curvatura, mapeado em tons de cinza.

- **Curvaturas principais**. As curvaturas mínima e máxima são armazenadas como componentes

Fig. 5.4: Visualização do valor absoluto da soma dos coeficientes do tensor de curvatura, em tons de cinza.

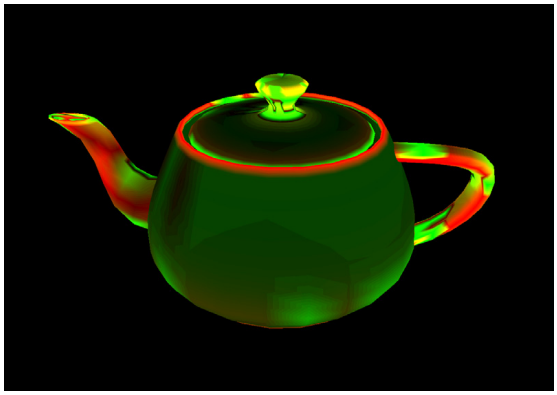


Fig. 5.5: Visualização das curvaturas principais, em cores falsas.

de cor do *pixel*. Internamente, o cálculo das curvaturas principais exige o cálculo dos coeficientes do tensor de curvatura. Em particular, as curvaturas principais são obtidas através do cálculo dos autovalores do tensor de curvatura.

Na aplicação, as curvaturas principais podem ser utilizadas para calcular outros tipos de curvatura, como a curvatura média $H = \frac{1}{2}(\kappa_{min} + \kappa_{max})$ e a curvatura Gaussiana $K = \kappa_{min}\kappa_{max}$.

Uma visualização dos atributos κ_1 e κ_2 obtidos em cada *pixel* é mostrado na figura 5.5. Nessa figura, κ_1 é mapeado na componente de cor verde, e κ_2 é mapeado na componente de cor vermelha.

- **Direções principais**. Cada componente dos vetores das direções principais é armazenada como uma componente de cor do *pixel*.

  Assim como na estimativa das curvaturas principais, o cálculo das direções principais utiliza

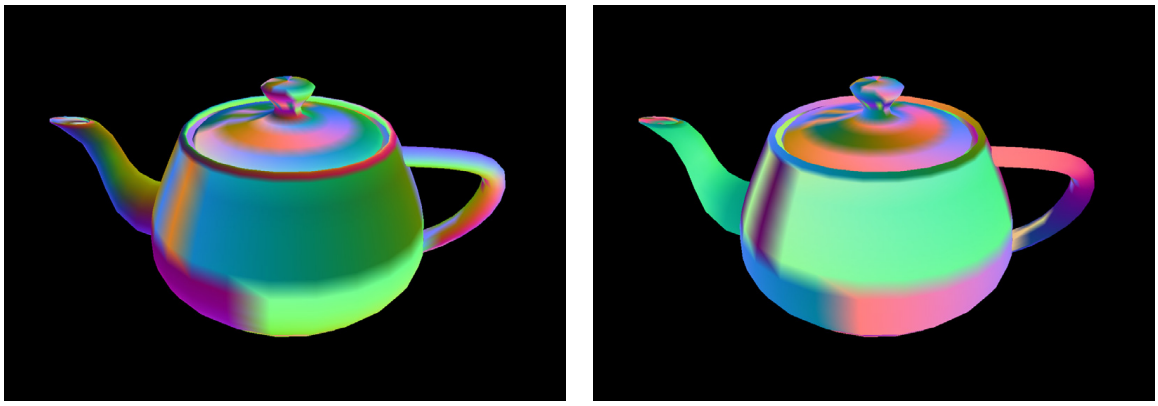


Fig. 5.6: Visualização dos vetores de direção principal mínima (esquerda) e máxima (direita), em cores falsas.

Fig. 5.7: Visualização do valor absoluto da soma dos coeficientes do tensor de derivada de curvatura, em tons de cinza.

internamente o cálculo dos coeficientes do tensor de curvatura. Em particular, as direções principais são obtidas através do cálculo dos autovetores do tensor de curvatura.

A figura 5.6 mostra duas imagens que ilustram, em cores falsas, os valores dos vetores de direção principal mínima e máxima, usando o mesmo mapeamento para valores RGB utilizado na figura 5.2.

- **Coeficientes do tensor de derivada de curvatura**. Cada coeficiente a, b, c, d do tensor $\mathbb{C}_s$ é armazenado numa componente de cor do *pixel* do *buffer* de renderização não visível. O cálculo desses coeficientes utiliza internamente o cálculo das curvaturas principais e direções principais.

  A figura 5.7 mostra uma visualização do valor absoluto da soma dos coeficientes do tensor de

derivada de curvatura, mapeado em tons de cinza.

- **Valor definido pela aplicação**. Além dos atributos geométricos descritos até aqui, nossa arquitetura suporta valores numéricos, definidos pela aplicação, para cada vértice de uma malha triangular indexada ou não indexada. Este atributo é fornecido na forma de um atributo adicional de cada vértice da geometria original (*e.g*, como um conjunto adicional de coordenadas de textura), com valor definido pela aplicação. Para geometria indexada, os valores definidos em um vértice são compartilhados por todas as faces adjacentes. Para geometria não indexada, cada vértice pode ter valores diferentes de acordo com a face que utiliza o vértice. A arquitetura suporta as duas versões de geometria (indexada e não indexada) ao mesmo tempo, dando assim mais flexibilidade para que a aplicação utilize atributos adicionais.

  Em geometria indexada é possível definir, para cada vértice, um valor constante que identifica univocamente aquele objeto. A arquitetura armazena esse valor no *buffer* de renderização não visível, para cada *pixel* do objeto. A aplicação pode ler esse valor para o *pixel* que contém o ponto de avaliação do cursor do dispositivo apontador e assim realizar uma tarefa de seleção. O mesmo princípio se aplica para identificar vértices. Neste caso, a aplicação atribui para cada vértice um valor que o define de forma unívoca.

  Em geometria não indexada, valores definidos pela aplicação podem ser utilizados para realizar tarefas tais como selecionar faces e obter as coordenadas baricêntricas de um ponto da face. Para selecionar faces, seus vértices devem conter um mesmo identificador que será então designado pela arquitetura para todos os *pixels* da face renderizada. Esse procedimento requer o uso de geometria não indexada, pois vértices coincidentes possuem identificadores diferentes para cada face adjacente.

  Coordenadas baricêntricas em uma face triangular podem ser obtidas a partir da atribuição de pesos a cada um dos três vértices da face. Por exemplo, considere, os pesos (s, t) da equação de coordenadas baricêntricas $V_1 + s(V_2 - V_1) + t(V_3 - V_1)$ de um triângulo (V_1, V_2, V_3). Os parâmetros s e t controlam a influência dos vértices V_1, V_2 e V_3 nas coordenadas. Na geometria original, os pares de valores $(0, 0)$, $(1, 0)$ e $(0, 1)$ podem ser designados pela aplicação, respectivamente, para cada conjunto de vértices V_1, V_2 e V_3. Durante a rasterização usando o modelo de interpolação *smooth* do OpenGL (também chamado de modelo *Gouraud* no Direct3D), esses valores serão interpolados linearmente para os *pixels* contidos dentro de cada face e poderão ser lidos pela aplicação para determinar as coordenadas baricêntricas do ponto coincidente com o *pixel*. Novamente, esse cálculo requer o uso de geometria não indexada, pois vértices coincidentes podem ter pesos diferentes para cada face adjacente.

  Uma visualização, em cores falsas, dos pesos da equação de coordenadas baricêntricas de-

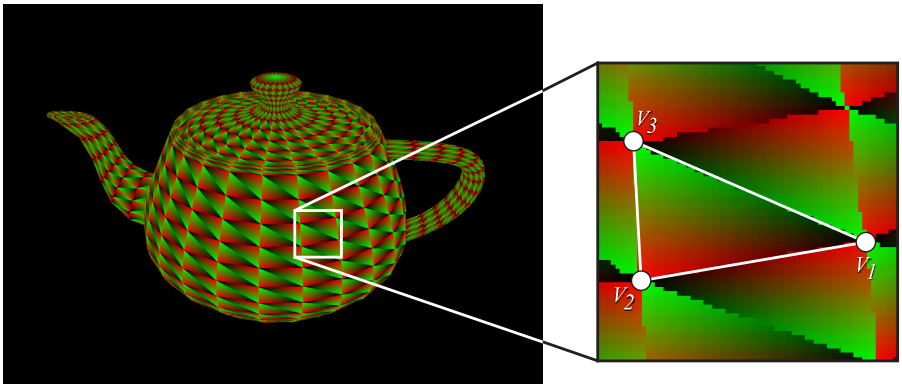


Fig. 5.8: Visualização, em cores falsas, de atributos definidos pela aplicação em geometria não indexada (pesos da equação de coordenadas baricêntricas).

signados como valores definidos pela aplicação para geometria não indexada é mostrada na figura 5.8. Os pesos s e t são mapeados nas componentes de cor vermelha e verde, respectivamente. O detalhe ampliado mostra a identificação dos vértices de uma das faces segundo a equação de coordenadas baricêntricas mostrada anteriormente.

A codificação de atributos geométricos em *buffers* de renderização não visíveis lembra o conceito de *geometry buffers* (*g-buffers*) freqüentemente utilizado na técnica de renderização conhecida como *deferred shading* [Saito and Takahashi, 1990a]. Nessa técnica, a avaliação da equação de iluminação é realizada numa etapa de pós-processamento dos dados do *frame buffer*, utilizando atributos geométricos previamente armazenados durante a rasterização em cada *pixel* dos *buffers* de renderização não visíveis. De fato, a arquitetura proposta utiliza o mesmo princípio de armazenar dados geométricos no *frame buffer*. Mas, em nossa proposta, eles são utilizados para manipulação direta 3D e podem ser estendidos também a atributos não geométricos. Além disso, o cálculo de atributos é necessário apenas para os *pixels* contidos na região de interesse em torno do cursor do dispositivo apontador – geralmente compreendendo apenas um *pixel* – em coordenadas do espaço da tela, e não para todos os *pixels* da imagem. Apesar de sua semelhança com o conceito de *g-buffers*, o uso da arquitetura proposta não impõe a utilização de *deferred shading* como técnica de visualização.

Sempre que as primitivas são modificadas na GPU por transformações que alteram as propriedades de geometria diferencial discreta dos modelos, essas propriedades precisam ser estimadas novamente para o novo modelo. Para esse propósito, a arquitetura inclui procedimentos para computar esses atributos inteiramente na GPU. Isso é feito após as modificações de atributos de vértices, garantindo assim resultados corretos em modelos geométricos deformados no processador de vértices, mesmo quando as funções de deformação não são conhecidas.

5.4 Fluxo de processamento

A arquitetura é construída sobre a camada de abstração de *hardware* gráfico definida segundo a API utilizada (*e.g.*, OpenGL ou Direct3D) e é visível ao programador como um conjunto adicional de funções de suporte a interação 3D. Internamente, a arquitetura explora a capacidade de utilizar a GPU como um processador de propósito geral e assim estimar, para cada vértices, os atributos de geometria diferencial descritos na seção 5.3, herdando as interfaces com o sistema de janelas suportado pelas atuais placas gráficas.

Uma vez que a arquitetura é centralizada em torno da idéia de utilizar o próprio fluxo programável de visualização para processar os atributos de geometria necessários para cada tarefa de manipulação direta 3D, as modificações de geometria em relação aos vértices e fragmentos podem ser levadas em consideração durante o processamento desses atributos. Evita-se a duplicação de dados uma vez que os *buffers* de vértices e de índices que definem a geometria utilizada para calcular os atributos podem ser os mesmos *buffers* utilizados para efetivamente visualizar o objeto.

Do ponto de vista da GPU, a arquitetura de interação é definida como uma série de procedimentos definidos em *shaders* de vértices e fragmentos, juntamente com definições de passos de renderização com saída direcionada a *buffers* de renderização não visíveis. Do ponto de vista da aplicação, a execução desses procedimentos é realizada no laço de renderização através da chamada de um comando `Render()` que processa os atributos de cada modelo e codifica esses atributos em *buffers* de renderização não visíveis, e um comando `Decode()` que retorna à aplicação os atributos codificados em cada *pixel*.

Uma visão geral dos procedimentos envolvidos na arquitetura de interação é ilustrado no fluxograma da figura 5.9. A arquitetura é composta de seis estágios responsáveis pelo processamento de atributos do modelo geométrico sob interação. Cinco desses estágios são executados no processador de vértices e fragmentos, e requerem até três passos de renderização:

- Primeiro passo de renderização: modificação de atributos de vértices antes do cálculo de atributos geométricos (estágio 1);
- Segundo passo de renderização: cálculo de atributos geométricos (estágio 2);
- Terceiro passo de renderização: modificação de atributos de vértices após o cálculo de atributos geométricos (estágio 3), modificação de atributos de fragmentos (estágio 4) e codificação de atributos de fragmentos (estágio 5);

O primeiro estágio executado na GPU recebe como entrada os atributos de vértices da geometria original (*i.e.*, o *buffer* de vértices e de índices utilizado na chamada de renderização) e os dados de

deformação utilizados para modificar os atributos de vértices e fragmentos na GPU (*e.g.*, texturas de deslocamento para utilizar em mapeamento de deslocamento, parâmetros de funções de deformação e matrizes de transformação). O segundo estágio estima as propriedades de geometria diferencial com base nos atributos e geometria modificados. Em um terceiro estágio, é possível alterar novamente os atributos dos vértices, que agora também contém as propriedades de geometria diferencial. Tal estágio pode ser utilizado por aplicações que necessitem dos atributos de geometria diferencial para realizar algum processamento na GPU. Em seguida, os atributos dos vértices são interpolados para cada fragmento de cada primitiva. No quarto estágio é possível modificar tais atributos interpolados. O quinto estágio codifica esses atributos de fragmentos nos *pixels* dos *buffers* de visualização não visíveis.

Além dessas etapas executadas na GPU, um sexto e último estágio é executado na CPU: o procedimento de *decodificação de atributos de fragmentos*. Neste estágio, a arquitetura fornece à aplicação o acesso aos atributos processados no *hardware* gráfico. A partir dessas informações, a tarefa de utilizar esses atributos para realizar a tarefa de interação e realimentação é de responsabilidade da aplicação.

Como indicado pelos estágios de decisão, as etapas de modificação de atributos de vértices, cálculo de atributos geométricos e modificação de atributos de fragmentos são opcionais. Elas são habilitadas pela aplicação apenas nos casos em que a deformação da geometria na GPU com relação aos vértices ou fragmentos é necessária. Se a geometria não é deformada na GPU, os últimos atributos geométricos calculados ficam armazenados em uma memória de textura para utilização apenas no estágio de codificação de atributos de fragmentos.

A seguir, na seção 5.4.1, detalhamos cada um dos seis estágios de processamento da arquitetura. Na seção 5.4.2 apresentamos o formato da seqüência de dados de entrada para execução da arquitetura e, na seção 5.4.3, o formato de saída dos dados computados.

5.4.1 Estágios de processamento

Os estágios de processamento são integrados ao laço de visualização da aplicação. Essa integração é mostrada na figura 5.10. Imediatamente antes de chamar a GPU para a renderização da geometria no laço de renderização, a aplicação invoca um comando `Render()` da arquitetura de interação que, internamente, dispara os três passos de renderização relacionados aos estágios 1, 2, 3, 4 e 5. Após a renderização da geometria para fins de visualização, a aplicação requisita o conteúdo dos *buffers* de renderização não visíveis (estágio 6) através da chamada do comando `Decode()`. Cada um desses seis estágios é detalhado a seguir. Por convenção, os procedimentos na figura 5.10 são exibidos como retângulos de bordas arredondadas, enquanto os repositórios de dados são exibidos como retângulos:

1. **Modificação de atributos de vértices**. Esse estágio é executado caso o modelo original

sofra alterações de seus atributos de vértices na GPU por funções de deformação definidas pela aplicação. Para cada modelo geométrico sob interação, esse procedimento é iniciado pela arquitetura através da chamada de uma função de chamada de retorno (*callback*) previamente definida pela aplicação através da interface entre a aplicação e a arquitetura de interação, e que contém o comando de renderização do modelo segundo a API utilizada (*e.g.*, `DrawIndexedPrimitive()` no Direct3D, ou `glDrawArrays()`, `glBegin()` e `glEnd()` no OpenGL). Tais funções recebem como parâmetro o *buffer* de vértices da geometria original, não deformada.

No processador de vértices, funções de deformação definidas pela aplicação modificam os atributos originais dos vértices e produzem a geometria final que será utilizada no cálculo da estimativa das propriedades geométricas do modelo. Para esse propósito, a geometria resultante ainda é produzida no sistema de coordenadas do espaço do objeto. Os atributos do modelo deformado são armazenados em texturas para uso posterior. Técnicas como mapeamento de deslocamento por vértice e *mesh skinning* são executadas neste estágio, da mesma forma que quaisquer outros *shaders* que modifiquem, com relação aos vértices, os atributos de geometria utilizados no cálculo de propriedades de geometria diferencial e bases do espaço de textura (atributos de posição 3D e coordenadas de textura).

2. **Cálculo de atributos geométricos**. Se a tarefa de interação demanda dados geométricos que não foram fornecidos como atributos dos vértices da geometria original (*e.g.*, vetor normal, curvaturas, direções principais, bases tangentes calculadas segundo o mapeamento de coordenadas de textura), ou que foram modificados no estágio de modificação de atributos de vértices, tais dados são calculados neste estágio de modo a levar em consideração a geometria deformada.

   A estimativa das propriedades de geometria diferencial é realizada para cada vértice do modelo, sempre após o estágio de modificação de atributos dos vértices. Assim como no estágio anterior, esse procedimento é disparado internamente pela arquitetura através da chamada de uma função de chamada de retorno contendo o comando de renderização do modelo com o *buffer* de vértices do modelo original. Para cada vértice, a GPU lê os atributos modificados do vértice através da amostragem de texturas de renderização utilizadas no estágio anterior e através da amostragem de texturas pré-computadas contendo dados de conectividade da geometria. Os resultados obtidos (atributos geométricos) são novamente gravados em texturas.

3. **Modificação de atributos de vértices**. A funcionalidade deste estágio é semelhante a do primeiro estágio. No *shader* de vértices, as texturas contendo os atributos da geometria deformada, obtidas dos estágios anteriores, são amostradas de modo a atualizar os atributos da geometria original. Desse modo, cada vértice possui atributos adicionais de geometria diferen-

cial calculados no estágio anterior. Tais atributos podem ser modificados arbitrariamente neste estágio antes de serem interpolados para cada fragmento durante a rasterização.

A inclusão deste estágio de modificação de atributos de vértices é justificada pela necessidade de dar mais flexibilidade aos tipos de transformações geométricas utilizadas no processador de vértices. Por exemplo, tal estágio é necessário para realizar tarefas tais como transformar bases tangentes recém calculadas, do sistema de coordenadas local do modelo para o sistema de coordenadas global. Como tais bases tangentes não são conhecidas no primeiro estágio, este tipo de processamento só pode ser realizado após as estimativas.

4. **Modificação de atributos de fragmentos**. Após projetar e rasterizar a geometria resultante dos estágios anteriores, os atributos interpolados dos fragmentos podem ser modificados. Durante a rasterização, a GPU realiza uma interpolação linear dos atributos dos vértices ao longo das primitivas. No *shader* de fragmentos correspondente a este estágio, os atributos interpolados podem ser modificados por uma função definida pela aplicação. Se essa função modificar atributos de posição e orientação de tal forma a requerer uma nova estimativa das propriedades de geometria diferencial, cabe à aplicação a tarefa de atualizar tais propriedades. Isso ocorre porque o estágio de estimativa de atributos geométricos é realizado apenas com relação aos vértices e não com relação aos fragmentos.

   Exemplos de processamento realizado neste estágio incluem as técnicas de mapeamento de detalhes 3D, execução de filtros de processamento de imagem e descarte de fragmentos. Além disso, texturas contendo atributos definidos pela aplicação podem ser amostradas neste estágio de modo a obter novos atributos, a critério da aplicação.

   Assim como nos estágios de deformação com relação aos vértices, este estágio é disparado internamente pela arquitetura através da chamada de uma função de chamada de retorno definida pela aplicação e que contém o comando de renderização do modelo segundo a API utilizada.

5. **Codificação de atributos**. Este procedimento é realizado no mesmo *shader* de fragmentos do estágio anterior. Os atributos definidos para cada fragmento a partir da interpolação dos fragmentos dos vértices, possivelmente modificados pelo estágio de modificação de atributos de fragmentos, são codificados como componentes de cor em *buffers* de renderização não visíveis. A funcionalidade de renderização simultânea em múltiplos *buffers* de renderização, existente nas GPUs atuais, é utilizada caso um único *buffer* de renderização não possua *bits* por *pixel* suficientes para armazenar todos os atributos.

6. **Decodificação de atributos**. Neste estágio, realizado na CPU e chamado no fim do laço de renderização, a arquitetura transfere o conteúdo dos *buffers* de renderização para a memória do

sistema e decodifica os atributos contidos em cada *pixel*. Esses atributos são então retornados à aplicação. Ao contrário dos estágios anteriores, que devem ser executadas uma vez para cada modelo geométrico sob interação, esse procedimento só precisa ser executado uma vez em cada iteração do laço de renderização.

Auxiliada pelo sistema de janelas, os atributos decodificados podem ser utilizados pela aplicação para concluir a tarefa de interação e fornecer alguma realimentação ao sistema, em geral um retorno visual (*e.g.*, atualizando a posição e orientação do cursor 3D). As ações de manipulação, definidas pela aplicação, também podem resultar em modificações da geometria original (*e.g.*, mudança da posição dos vértices ou faces), modificações da conectividade da geometria (*e.g.*, em uma operação de recorte e costura de superfícies), modificações do conteúdo das texturas utilizadas pelo modelo (*e.g.*, decorrentes de uma tarefa de pintura 3D) ou modificações dos parâmetros das funções de deformação. Em tais casos, a aplicação é responsável por informar essas mudanças à arquitetura de interação de acordo com a interface de entrada descrita na seção 5.4.2.

5.4.2 Interface de entrada (CPU-GPU)

Para iniciar o processamento de atributos na GPU, a arquitetura requer que a aplicação forneça a seguinte seqüência de dados de entrada para cada modelo geométrico a ser processado:

- *Buffer* de vértices da geometria original, idêntico ao *buffer* de vértices utilizado na renderização do modelo para visualização, porém com um atributo adicional contendo o índice do vértice no *buffer*. Esse índice é utilizado pela arquitetura durante os estágios de processamento 1, 2 e 3 para calcular o índice 2D de acesso das texturas que contém dados da geometria e topologia do modelo. O procedimento de conversão do índice 1D para 2D é detalhado na seção 4.1.1.

  Cada vértice deve conter também os atributos definidos na aplicação (*atributos de entrada*) necessários para calcular os atributos que serão processados na GPU e armazenados nos *buffers* de renderização não visíveis (*atributos de saída*). Esses dados podem ser fornecidos através do comando `SetVertexBuffer()` que recebe o ponteiro do *buffer* de vértices. Internamente, a arquitetura utiliza esse *buffer* de vértices para criar as texturas que serão utilizadas nos estágios 1, 2 e 3.

  Os atributos de entrada exigidos para o cálculo de cada atributo de saída são citados na seção 5.3. Por exemplo, o cálculo do valor de profundidade requer apenas a existência da posição 3D como atributo de entrada de cada vértice no *buffer* de vértices. O cálculo dos pesos das coordenadas baricêntricas requer, como atributo de entrada, os valores 0, 1 e 2 designados para cada um dos

três vértices de cada face. Esses valores devem ser definidos pela aplicação e acrescentados ao *buffer* de vértices da geometria original.

- *Buffer* de índices da geometria original, idêntico ao *buffer* de índices aos vértices utilizado na renderização do modelo para visualização. Esses dados podem ser fornecidos através do comando `SetIndexBuffer()` que recebe o ponteiro do *buffer* de índices. Cada seqüência de três índices do *buffer* deve corresponder a um triângulo. Internamente, a arquitetura utiliza esses dados para determinar a vizinhança de 1-anel de cada vértice, e armazenar esses dados em texturas que serão utilizadas no estágio 2.

- Relação dos atributos que devem ser processados pela GPU e armazenados para cada *pixel* do modelo renderizado. Essa relação pode ser qualquer combinação dos atributos citados na seção 5.3, e pode ser fornecida à arquitetura através do comando `SetAttributes()`.

- Mapeamento da semântica dos atributos de entrada, entre a API gráfica e a arquitetura de interação. Nas APIs gráficas, os atributos de cada vértice da geometria original possuem uma semântica pré-definida, necessária para a correta interpretação de cada atributo na entrada do processador de vértices ou no fluxo de função fixa de visualização. Por exemplo, um vértice pode ser composto de valores cuja semântica, do ponto de vista da API gráfica, é uma posição 3D, coordenadas de textura 2D, um valor de cor difusa ou tamanho do ponto. Do ponto de vista da arquitetura proposta, tais atributos podem ter outra semântica. Por exemplo, valores numéricos designados aos vértices com semântica de *tamanho do ponto* (do ponto de vista da API), podem ser interpretados pela arquitetura de interação como um *identificador de vértice*, ou como um valor genérico com semântica definida pela aplicação.

  Esse mapeamento pode ser fornecido à arquitetura através do comando `BindSemantics()` que recebe duas listas cujos valores são mapeados biunivocamente. A primeira lista contém as semânticas dos atributos dos vértices segundo a API gráfica. A segunda lista contém as semânticas dos atributos dos vértices segundo a arquitetura de interação.

- Coordenadas da região de interesse. A extensão espacial da codificação dos atributos das geometrias processadas é configurada de acordo com a definição de uma região retangular de interesse, especificada em coordenadas da janela. Para tarefas simples de interação como seleção e posicionamento restrito à superfície mais próxima, essa região pode envolver apenas o *pixel* apontado pelo cursor 2D controlado pelo dispositivo apontador. Nessas tarefas, as primitivas sob interação são apenas as primitivas apontadas pelo cursor 2D. Primitivas situadas fora da região de interesse são descartadas e o processamento de fragmentos é reduzido a fragmentos dentro dessa região. Em tarefas de posicionamento com restrição a vértices, arestas, ou caracte-

rísticas da imagem, a região de interesse pode determinar a extensão do "campo de gravidade", pois a aplicação pode fazer com que o cursor 2D seja atraído à característica mais próxima dentro da região de interesse.

Esses dados podem ser fornecidos à arquitetura através do comando `SetROI()` que recebe como parâmetros as coordenadas 2D do canto superior esquerdo e canto inferior direito da região de interesse.

- *Shaders* de deformação de vértices, definidos pela aplicação. Aplicam-se apenas quando há modificação dos atributos de vértices de entrada, ou dos atributos produzidos pelo estágio de cálculo de atributos geométricos. Quando utilizado no primeiro estágio, o *shader* de deformação de vértices recebe como parâmetro de entrada os atributos da geometria original (com os tipos de dados e semântica definidas anteriormente) e produz como saída a mesma lista de atributos, mas com valores possivelmente alterados segundo o processamento definido pela aplicação. A entrada do *shader* de deformação de vértices utilizado no terceiro estágio inclui não só todos os parâmetros do primeiro estágio, como também os novos atributos correspondentes às propriedades de geometria diferencial estimadas no segundo estágio.

  Esses dados podem ser fornecidos à arquitetura através de dois comandos, `SetPreVertexDeform()` e `SetPostVertexDeform()`, que recebem como parâmetro uma *string* de texto contendo o código do *shader* de modificação de atributos de vértices. O comando `SetPreVertexDeform()` recebe o *shader* a ser executado antes da etapa de estimativa de propriedades de geometria diferencial. O comando `SetPostVertexDeform()` recebe o *shader* a ser executado após a etapa de estimativa de propriedades de geometria diferencial.

- *Shader* de deformação de fragmentos, definido pela aplicação. Aplica-se apenas quando há modificação dos atributos de entrada interpolados ao longo de cada fragmento produzido durante a rasterização. A entrada do *shader* de deformação de fragmentos recebe como parâmetros os atributos de entrada (com os tipos de dados e semântica definidas anteriormente) interpolados linearmente pelo rasterizador, e produz como saída a mesma lista de atributos, mas com valores possivelmente alterados segundo o processamento definido pela aplicação.

  Esses dados podem ser fornecidos à arquitetura através do comando `SetPixelDeform()` que recebe como parâmetro uma *string* de texto contendo o código do *shader* de modificação de atributos de fragmentos.

- Função de chamada de retorno a ser utilizada pela arquitetura quando a atualização dos atributos de vértices (estágio 1) e estimativa das propriedades de geometria diferencial (estágio 2)

forem realizadas. Tal função deve conter a chamada do comando de renderização das primitivas construídas com os *buffers* de vértices da geometria original indexada.

A função de chamada de retorno não deve chamar nenhuma função de modificação do estado de renderização relacionado ao preenchimento das primitivas, como modificação do estado `GL_POLYGON_MODE` do OpenGL ou `D3DFILLMODE` do Direct3D, pois isso pode interferir na execução dos *shaders* internos chamados pela arquitetura. Também não é permitido modificar os *shaders* dentro desta função, mas é possível atribuir valores aos registradores constantes e texturas utilizadas pelo *shader* de modificação de atributos de vértices do estágio 1.

Essa função de chamada de retorno pode ser informada à arquitetura através do comando `SetUpdateCallback()` que recebe como parâmetro o ponteiro da função.

- Função de chamada de retorno a ser utilizada pela arquitetura quando os estágios 3, 4 e 5 forem executados. Assim como na função de chamada de retorno dos estágios 1 e 2, tal função deve conter a chamada do comando de renderização das primitivas construídas com os *buffers* de vértices da geometria original. Essas funções podem modificar quaisquer estados de renderização, com exceção da modificação dos *shaders* que estão sendo utilizados, pois isto é definido internamente pela arquitetura.

  Duas funções de chamada de retorno podem ser utilizadas: uma para renderizar geometria indexada e outra para renderizar geometria não indexada. Essas funções podem ser informadas à arquitetura através de um comando `SetRenderCallback()` que recebe como parâmetros os ponteiros das funções.

A definição das primitivas construídas com os atributos de vértices e a decisão sobre o uso de geometria indexada são obtidas diretamente através do uso das funções de renderização da API nas funções de chamada de retorno. Outros dados de entrada utilizados na renderização dos atributos, tais como os registradores constantes contendo os parâmetros das funções de deformação, e conjuntos de texturas contendo dados para modificação dos atributos de fragmentos, têm seu uso definido pela própria aplicação dentro dos *shaders* de deformação de vértices ou fragmentos.

5.4.3 Interface de saída (GPU-CPU)

Após o estágio de modificação de atributos de fragmentos, os atributos de fragmentos são codificados no formato de cor de um ou mais *buffers* de renderização não visíveis.

A forma como os atributos serão codificados como cores pode ser modificada de acordo com a precisão numérica dos resultados requerida. Por exemplo, cada valor pode ser armazenado em uma componente de cor de 32 bits em formato de ponto flutuante. Assim, as três componentes de um vetor

normal serão armazenadas em três componentes distintas de cor, cada uma com 32 bits de precisão. Tal procedimento é utilizado em nossa implementação da arquitetura. Alternativamente, porém, esses valores podem ser armazenados com menor precisão em componentes de cor de 16 e 8 bits sem sinal, diminuindo assim o consumo de memória de vídeo. Considerando vetores normalizados (portanto, com valores dos seus componentes situados entre -1 e 1), cada componente p do vetor pode ser convertido em uma cor c (com valores entre 0 e 1) através da fórmula $c = (p + 1)/2$ e decodificado na CPU usando $p = 2c - 1$.

O procedimento de decodificação dos atributos é executado pela aplicação através da chamada do comando `Decode()`. Internamente, tal comando lê o conteúdo da região de interesse de cada um dos *buffers* de renderização utilizados na codificação, decodifica os atributos de fragmentos contidos neles e retorna-os à aplicação.

5.5 Procedimento de uso

A interface de programação da arquitetura proposta é discutida detalhadamente no apêndice C. Nesta seção, descrevemos os passos necessários para implementar uma tarefa de manipulação direta 3D segundo tal interface. Para inicializar a arquitetura, os seguintes procedimentos são realizados através de chamadas de comandos da arquitetura (a ordem de execução dos comandos não é relevante):

- Criação do *buffer* de vértices de cada modelo. Cada vértice inclui necessariamente como atributo adicional o índice do vértice no *buffer*. A aplicação utiliza o comando `SetVertexBuffer()` para informar esse *buffer* à arquitetura.

  Dependendo da tarefa de interação 3D desejada, atributos adicionais devem ser incluídos. Consideremos, por exemplo, uma tarefa de seleção em uma cena 3D constituída de n modelos. A seleção consistirá em identificar qual modelo está sendo apontado pelo cursor 2D. Segundo o paradigma de armazenar os atributos no espaço da imagem, isto pode ser obtido se codificarmos os identificadores dos modelos em cada *pixel* dos modelos renderizados. Para tanto, o *buffer* de vértices de cada modelo deve conter como atributo adicional o identificador do modelo, *i.e.*, um valor entre 1 e n, onde n é o número de modelos da cena 3D.

- Criação do *buffer* de índices de cada modelo, informado à arquitetura através do comando `SetIndexBuffer()`.

- Definição dos atributos que serão codificados para cada *pixel* dos modelos renderizados, usando o comando `SetAttributes()`. Se esses atributos incluírem propriedades de geometria diferencial, o estágio 2 será habilitado.

No exemplo de uma tarefa de seleção, esse comando será utilizado para informar que somente um "atributo definido pela aplicação" será codificado. Tal atributo é o identificador do modelo, definido como um atributo adicional de cada vértice do *buffer* de vértices.

- Definição do mapeamento da semântica dos atributos. Para isso a aplicação chama o comando `BindSemantics()` informando necessariamente qual atributo do *buffer* de vértices contém o índice do vértice.

  No exemplo de uma tarefa de seleção, esse comando deve ser utilizado para indicar qual atributo do *buffer* de vértices contém o "atributo definido pela aplicação." Neste caso, é o atributo que contém o identificador do modelo.

- Definição dos *shaders* de modificação de atributos de vértices e fragmentos, caso eles se apliquem aos modelos utilizados. Isto é feito através dos comandos `SetPreVertexDeform()`, `SetPostVertexDeform()` e `SetPixelDeform()`. Se o modelo não sofrer qualquer deformação, esses comandos não serão utilizados, e os estágios 1, 3 e 4 da arquitetura não serão habilitados.

- Definição das funções de chamada de retorno contendo as chamadas de renderização do modelo (*e.g.*, `glDrawArrays()`), empregando o *buffer* de vértices utilizado em `SetVertexBuffer()`. Os comandos `SetUpdateCallback()` e `SetRenderCallback()` são utilizados para informar à arquitetura tais funções de chamada de retorno.

No laço de renderização, os seguintes procedimentos são executados:

- Definição da região de interesse, através do comando `SetROI()`.

  Em uma tarefa de seleção, a região de interesse será definida pelas coordenadas da posição atual do cursor 2D. Assim, somente o *pixel* apontado pelo cursor conterá os atributos processados pela arquitetura.

- Execução do fluxo de processamento da arquitetura, através da execução dos comandos `Render()` e `Decode()` (nesta ordem). Através desses comandos a arquitetura executa internamente os estágios 1 a 6. Os dados obtidos desses procedimentos serão os atributos armazenados para cada *pixel* dos modelos renderizados dentro da região da tela especificada pela região de interesse.

  No exemplo de uma tarefa de seleção, o comando `Decode()` retornará apenas um atributo codificado para o *pixel* dentro da região de interesse: o "atributo definido pela aplicação", *i.e.*, o identificador do modelo segundo o mapeamento da semântica dos atributos. Uma vez que a

região de interesse contém o *pixel* apontado pelo cursor 2D, o atributo retornado é, de fato, o identificador do modelo apontado pelo cursor.

A seguir mostramos uma listagem de código exemplificando a aplicação deste procedimento de uso para a implementação de uma tarefa simples de seleção. Os comandos utilizados seguem a interface de programação descrita no apêndice C e utilizam a linguagem C++ com a API OpenGL e a biblioteca utilitária GLUT. Por simplicidade, neste exemplo a geometria é apenas um triângulo, e os *shaders* de modificação de atributos de vértices e fragmentos não modificam esses atributos (figura 5.11). A implementação deste procedimento de seleção em uma cena mais complexa é descrita na seção 6.2.

Ressaltamos que, comparando com a programação convencional, foi necessário fazer somente as seguintes modificações:

- Instanciar a arquitetura de interação e inicializar o ambiente junto com as funções de inicialização do OpenGL na função `Init()`.
- Chamar, na função de renderização `Display()`, o comando de execução do fluxo de processamento da arquitetura e o comando de decodificação dos atributos calculados.
- Definir a região de interesse em relação à posição do cursor do dispositivo apontador 2D.

```
#include "CIntManager.h"

#include <GL/glut.h>
#include <stdlib.h>

// Tamanho da região de interesse
#define ROIW 1
#define ROIH 1

CIntManager *g_pIntManager = NULL;
float *g_pROI[1]; // Array com o conteúdo da região de interesse

float g_fNear =  1.0; float g_fFar  = 20.0;

// Função de chamada de retorno para atualizar e renderizar a geometria
void UpdateRenderCallBack( CIntObj *pObj, void *pUserData ) {
  glPushMatrix( );
  glBegin( GL_TRIANGLES );
```

```
    glColor4f( 0.0, 0.0, 0.0, 0.0 ); // ID do vértice
    glTexCoord4i( 1, 1, 1, 1 ); // ID do modelo
    glVertex3f( -2.0, -2.0, 0.0 );

    glColor4f( 1.0, 1.0, 1.0, 1.0 ); // ID do vértice
    glTexCoord4i( 1, 1, 1, 1 ); // ID do modelo
    glVertex3f( 2.0, -1.0, 0.0 );

    glColor4f( 2.0, 2.0, 2.0, 2.0 ); // ID do vértice
    glTexCoord4i( 1, 1, 1, 1 ); // ID do modelo
    glVertex3f( 0.0, 2.0, 0.0 );
  glEnd( );
  glPopMatrix( );
}

void Init( ) {
  // Cria instância da arquitetura
  g_pIntManager = new CIntManager( );

  // Informa os atributos desejados
  ATTRIBUTE m_stAttribute[] =
  {
    ATTTYPE_USERDEFI, // Identificador do modelo
    ATTTYPE_END
  };
  g_pIntManager->SetAttributes( m_stAttribute );

  // Informa as ligações semânticas
  SEMANTICBINDING stBinding[] =
  {
    // Identificador do vértice está no valor de cor
    { VSSEMANTIC_COLOR0, INTSEMANTIC_VERTEXID  },
    // Identificador do modelo está no conjunto de
    // coordenadas de textura
    { VSSEMANTIC_TEXCOORD0, INTSEMANTIC_USERDEFI  },
    SEMANTIC_END
  };
  g_pIntManager->BindSemantics( stBinding );
```

```
// Cria instância de CIntObj
CIntObj *pMyObj = NULL;
g_pIntManager->CreateObject( &pMyObj, NULL, NULL );

// Informa as funções de chamada de retorno
pMyObj->SetUpdateCallback( UpdateRenderCallBack );
pMyObj->SetRenderCallback( UpdateRenderCallBack, NULL );

// Informa shader de modificação de atributos de vértices
// antes do cálculo de atributos geométricos (estágio 1)
static char *szVSPreDeform = {
  "VSDEFORM fPreVSDeform( VSDEFORM In )                    \n"
  "{                                                       \n"
  "  VSDEFORM Out = In;                                    \n"
  "  // Funções de modificação dos atributos de Out vão aqui \n"
  "  return Out;                                           \n"
  "}                                                       \n"
};
pMyObj->SetPreVertexDeform( szVSPreDeform );

// Informa shader de modificação de atributos de vértices
// depois do cálculo de atributos geométricos (estágio 2)
static char *szVSPostDeform = {
  "PSDEFORM fPostVSDeform( PSDEFORM In )                   \n"
  "{                                                       \n"
  "  PSDEFORM Out = In;                                    \n"
  "  // Funções de modificação dos atributos de Out vão aqui \n"
  "  return Out;                                           \n"
  "}                                                       \n"
};
pMyObj->SetPostVertexDeform( szVSPostDeform );

// Informa shader de modificação de atributos de fragmentos
static char *szPSDeform = {
  "PSDEFORM fPSDeform( PSDEFORM In )                       \n"
  "{                                                       \n"
  "  PSDEFORM Out = In;                                    \n"
```

```
    "  // Funções de modificação dos atributos de Out vão aqui \n"
    "  return Out;                                              \n"
    "}                                                          \n"
  };
  pMyObj->SetPixelDeform( szPSDeform );

  // Informa buffer de vértices e de índices
  float pVtPos[9] = { -2.0, -2.0, 0.0,
                       2.0, -1.0, 0.0,
                       0.0,  2.0, 0.0 };
  int pIdx[3] = { 0, 1, 2 };

  pMyObj->SetVertexBuffer( pVtPos, NULL, NULL, 3 );
  pMyObj->SetIndexBuffer( pIdx, 1 );

  glEnable( GL_DEPTH_TEST );
  glClearColor( 0.0, 0.0, 0.0, 0.0 );

  // Aloca memória para a região de interesse
  g_pROI[0] = new float[ROIW*ROIH*4]; // 1 pixel com 1 atributo RGBA

  g_pIntManager->SetROI( 0, 0, ROIW-1, ROIH-1 );
}

void Display( ) {
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

  // Renderiza modelo para visualização
  RenderCallBack( NULL, 0, NULL );

  // Executa o fluxo de processamento da arquitetura e
  // renderiza o modelo nos buffers de visualização não visíveis
  g_pIntManager->Render( );

  // Decodifica os atributos
  g_pIntManager->Decode( g_pROI );

  glutSwapBuffers( );
```

```
  if( g_pROI[0][0] )
    printf( "Modelo selecionado!\n" );
}

void Reshape( int iW, int iH ) {
  glViewport( 0, 0, (GLsizei) iW, (GLsizei) iH );
  glMatrixMode( GL_PROJECTION );
  glLoadIdentity( );
  gluPerspective( 60.0, (GLfloat) iW/(GLfloat) iH, g_fNear, g_fFar );
  glMatrixMode( GL_MODELVIEW );
  glLoadIdentity( );
  gluLookAt( 0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0 );

  // Distância dos planos de recorte para a região de interesse
  g_pIntManager->SetZNear( g_fNear );
  g_pIntManager->SetZFar ( g_fFar  );
}

void Mouse( int iX, int iY ) {
  // Região de interesse envolve apenas
  // o pixel apontado pelo cursor
  g_pIntManager->SetROI( iX, iY, iX+ROIW-1, iY+ROIH-1 );
}

int main( int argc, char** argv ) {
  glutInit( &argc, argv );
  glutInitDisplayMode( GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA );
  glutInitWindowSize( 600, 600 );
  glutInitWindowPosition( 0, 0 );
  glutCreateWindow( argv[0] );

  Init( );
  glutDisplayFunc( Display );
  glutIdleFunc( Display );
  glutReshapeFunc( Reshape );
  glutPassiveMotionFunc( Mouse );
  glutMainLoop( );
```

```
  return 0;
}
```

5.6 Considerações finais

A arquitetura proposta fornece à aplicação uma plataforma comum para implementar diferentes tarefas de manipulação direta 3D capazes de utilizar o poder computacional das GPUs. Em particular, as GPUs podem ser utilizadas para calcular de forma eficiente os atributos necessários para interação, e ao mesmo tempo levar em consideração as possíveis alterações desses atributos durante o fluxo de renderização.

A interface de programação é baseada na arquitetura de GPUs compatíveis com o modelo de *shader* 3.0, e pode ser revista conforme a evolução do *hardware* gráfico. Por exemplo, em GPUs compatíveis com o modelo de *shader* 4.0, a introdução do processador de geometria com a de possibilidade criar primitivas na GPU implica a revisão da interface de programação para contemplar tais alterações. No entanto, acreditamos que a filosofia da arquitetura proposta (figuras 5.9 e 5.10) permanece válida enquanto a arquitetura das placas seguir o paradigma do fluxo programável de renderização mostrado na figura 2.3.

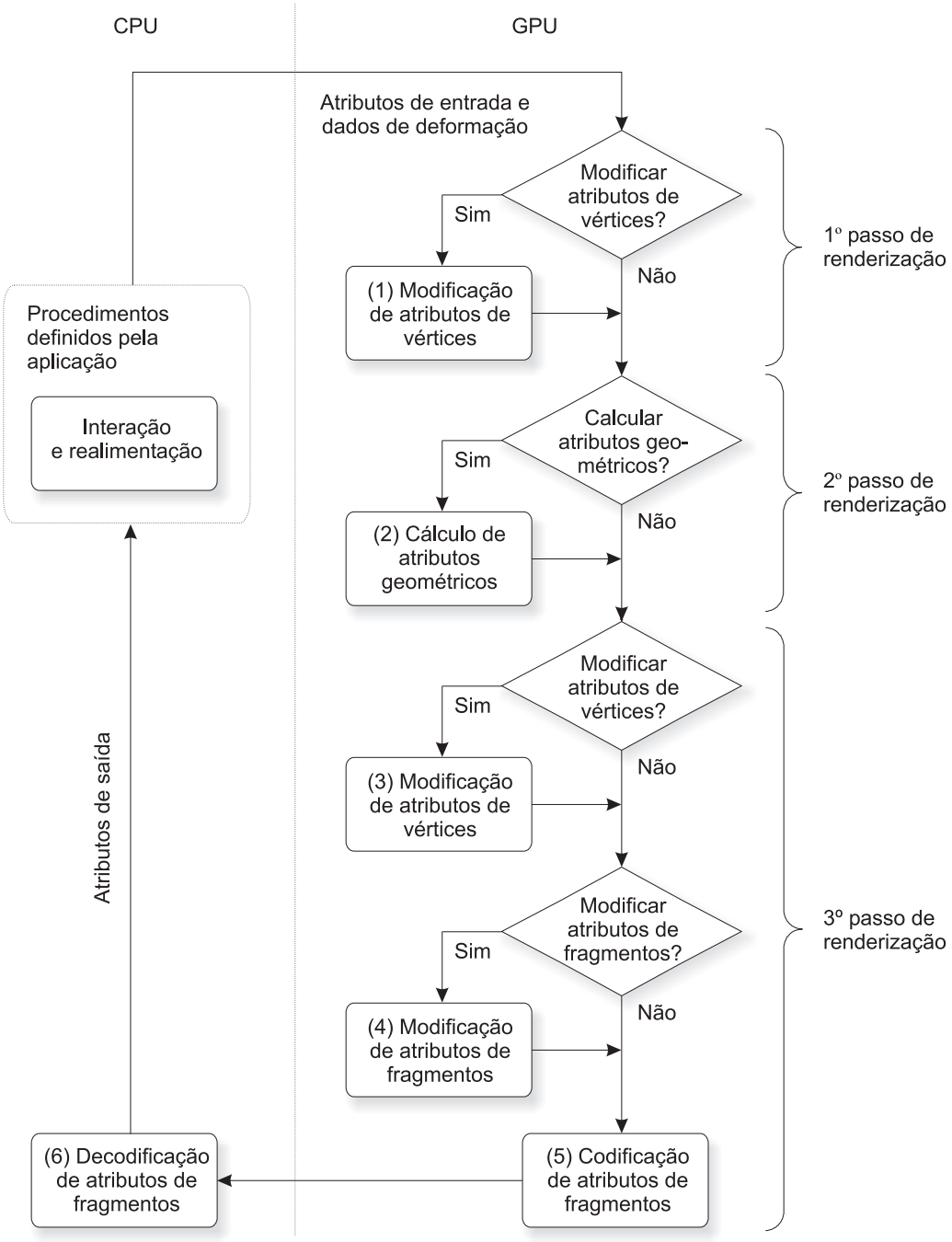


Fig. 5.9: Visão geral dos procedimentos envolvidos na arquitetura de interação.

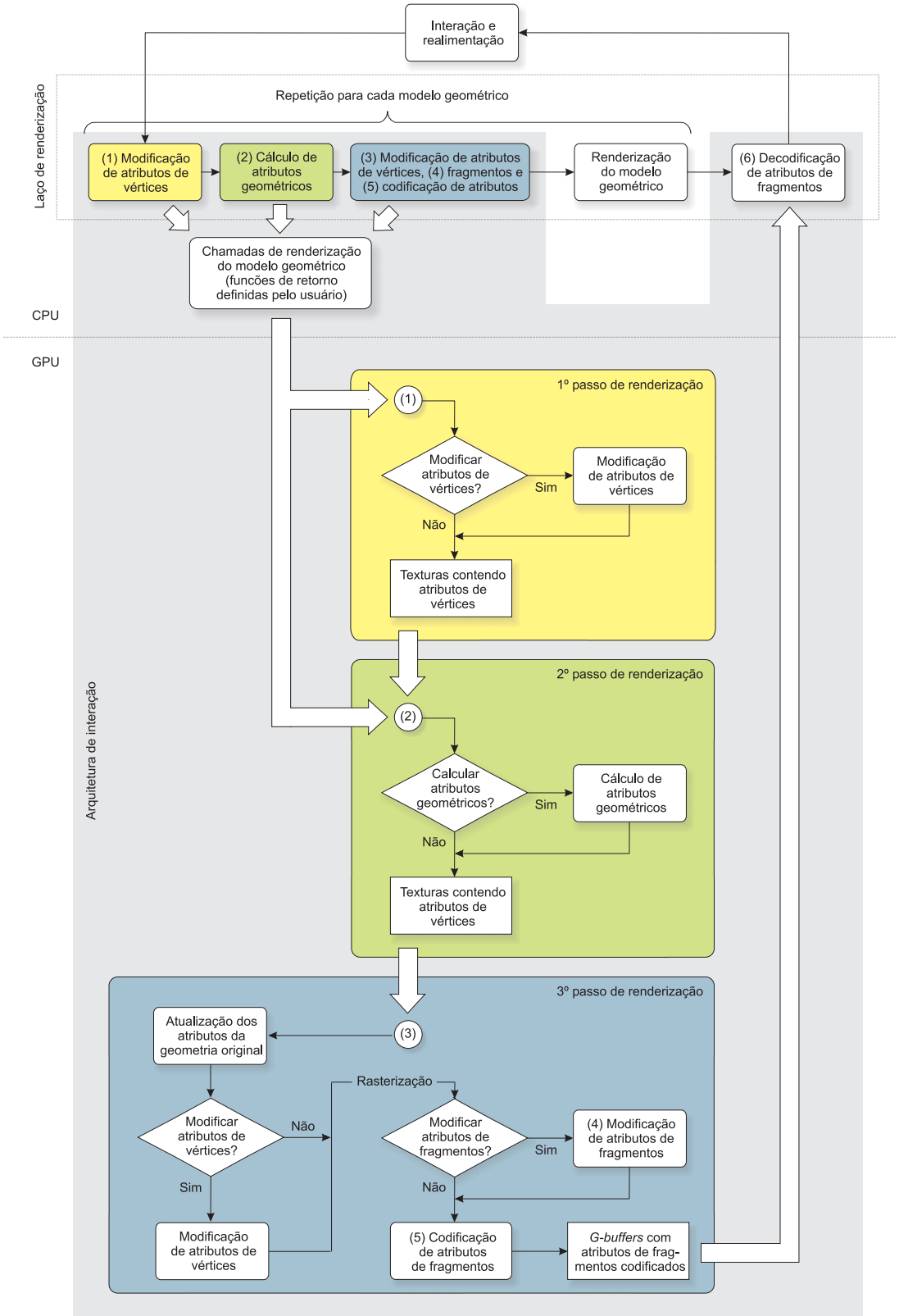


Fig. 5.10: Integração entre os estágios de processamento da arquitetura (região sombreada em cinza) e o laço de visualização na aplicação.

Fig. 5.11: Janela da aplicação de seleção usando a arquitetura proposta.

Capítulo 6

Resultados

Neste capítulo apresentamos resultados de testes de desempenho e aplicações da arquitetura proposta que permitem validar as hipóteses defendidas nesta tese:

- O uso de atributos geométricos (propriedades de geometria diferencial) e não geométricos (valores definidos pela aplicação) armazenados em cada *pixel* dos modelos renderizados é suficiente para permitir a implementação de tarefas de manipulação direta 3D com o uso de dispositivos apontadores 2D.
- É factível integrar às atuais GPUs mecanismos de interações capazes de levar em consideração os modelos deformados no fluxo programável do *hardware* gráfico.

De modo a demonstrar a eficiência do uso da nossa arquitetura em *hardware* gráfico atual, apresentamos resultados de testes de desempenho obtidos da execução de diferentes tarefas de manipulação direta 3D baseadas no uso da arquitetura implementada segundo a biblioteca de funções apresentada no apêndice C. Esses resultados são apresentados e discutidos na seção 6.1.

Na seção 6.2 mostramos exemplos de uso da arquitetura proposta para implementar tarefas compostas de manipulação direta 3D obtidas a partir das tarefas básicas de manipulação direta discutidas no capítulo 3. Para cada tarefa implementada, detalhamos os comandos da biblioteca utilizados em cada tarefa, mostrando a simplicidade tanto na implementação quanto no tratamento de modelos que sofrem modificações de seus atributos de vértices e fragmentos durante o processamento na GPU.

6.1 Testes de desempenho

Como primeiro teste de desempenho, comparamos o tempo de processamento necessário para realizar uma tarefa básica de interação usando o método tradicional de *ray picking* e nossa arquitetura

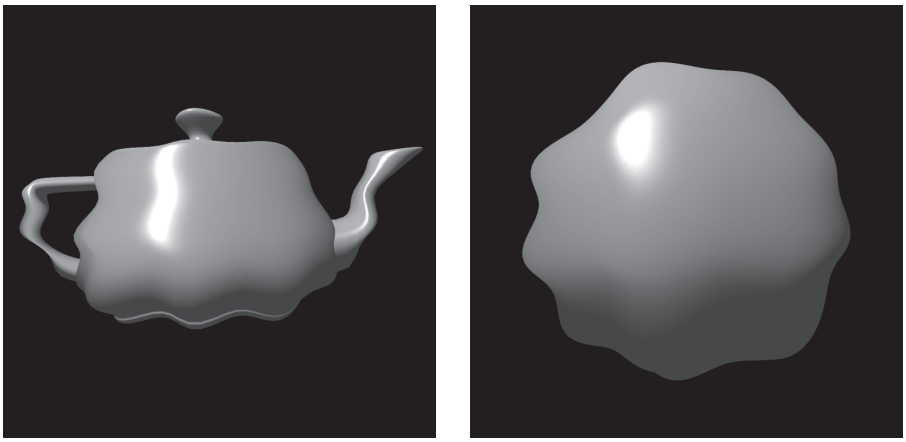


Fig. 6.1: Modelos utilizados no teste de desempenho entre o método de *ray picking* e a arquitetura proposta.

de interação. Esta tarefa de interação consistiu em realizar o posicionamento restrito de um cursor 3D a um ponto de uma superfície apontada pelo cursor 2D, alinhando o cursor 3D com relação ao vetor normal da superfície no ponto de restrição. Os testes foram conduzidos em um computador computador AMD Athlon 64 3500+ de 2.2GHz e 2GB RAM, equipado com uma placa gráfica NVIDIA GeForce 8800 GTX com 768MB. A geometria utilizada foi deformada em cada quadro de exibição, de modo que os atributos geométricos necessários para realizar a interação precisaram ser calculados em tempo de execução, em todos os quadros. No método de *ray picking*, estes cálculos foram feitos na CPU. Na arquitetura proposta, as deformações foram realizadas diretamente no processador de vértices, no estágio de modificação de atributos de vértices (estágio 1 da arquitetura segundo as figuras 5.9 e 5.10).

Na implementação do método de *ray picking*, utilizamos a função `D3DXIntersect()` fornecida pela biblioteca D3DX da API Direct3D [Microsoft, 2006]. Esta função é utilizada para testar a interseção entre um raio e um modelo geométrico e retornar atributos geométricos nos pontos de interseção. Ao utilizar a biblioteca D3DX, as modificações dos atributos dos vértices foram realizadas na CPU. Além disso, não utilizamos modificações de atributos de fragmentos, pois a função de cálculo de interseção do D3DX é incapaz de trabalhar com deformações da geometria em relação aos fragmentos produzidos na rasterização. Isso ocorre porque os testes de interseção são realizados somente sobre a malha triangular da geometria.

Os modelos utilizados neste teste de desempenho foram uma chaleira e uma esfera deformadas por funções senoidais em tempo real e exibidos como listas de triângulos não indexados. Uma visualização desses modelos com a deformação aplicada é mostrada na figura 6.1. A esfera foi composta de 19.802 vértices. A chaleira teve duas versões de refinamento de malha. A geometria menos refi-

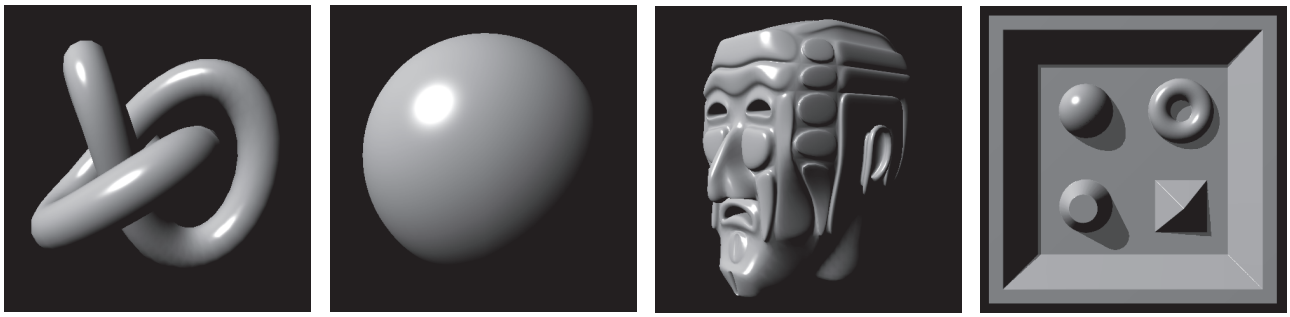
Modelo	Somente visualização	*Ray picking*	Nossa arquitetura
Chaleira (1.178 vértices)	0,32	76,49	1,066
Esfera (19.802 vértices)	0,32	1351,56	4,967
Chaleira (51.362 vértices)	0,54	3478,27	11,383

Tab. 6.1: Tempo médio de renderização (em milissegundos) de modelos deformados por funções senoidais e utilizados para interação segundo o método de *ray picking* e nossa arquitetura.

nada foi composta de 1.178 vértices, e a mais refinada foi composta de 51.362 vértices. Os atributos processados neste teste foram os atributos de valor de profundidade (para obtenção da posição 3D) e vetor normal. Em nossa arquitetura, o vetor normal foi calculado na GPU. No método de *ray picking*, este processamento foi realizado na CPU. Os resultados são mostrados na tabela 6.1 e contêm o tempo total de renderização de cada modelo, em milissegundos. A coluna "Somente visualização" mostra o tempo de renderização dos modelos apenas para visualização, *i.e.*, sem processamento dos atributos necessários para interação, mas ainda assim utilizando as funções de deformação da posição dos vértices. A coluna "*Ray picking*" mostra o tempo de renderização juntamente com o tempo de execução do algoritmo de seleção da biblioteca D3DX. A coluna "Nossa arquitetura" mostra o tempo de renderização quando nossa arquitetura foi utilizada.

Para os três modelos, observamos que o desempenho obtido com nossa arquitetura foi substancialmente melhor do que aquele obtido com o método de *ray picking* implementado através das funções da biblioteca D3DX. Isso é esperado, pois o gargalo principal do método baseado na função da biblioteca D3DX está em sua limitação de usar processamento apenas na CPU, e na necessidade de recalcular, para cada quadro de exibição, o *buffer* de vértices do modelo deformado com base na geometria original. Nossa arquitetura produz resultados mais eficientes em razão do maior poder computacional das GPUs, seja para acessar os dados geométricos, seja para renderizar a geometria e assim obter os atributos equivalentes aos obtidos dos testes de interseção entre a geometria e o raio de seleção. Isso se torna mais evidente quando se aumenta a complexidade do modelo tratado. Em particular, as funções de deformação são executadas na GPU, assim como a estimativa dos vetores normais. O *buffer* de vértices utilizado é sempre aquele da geometria original, e pode então ser armazenado em memória de vídeo local para ser acessado com maior eficiência.

É razoável supor que o desempenho de nossa arquitetura depende da quantidade e dos tipos de atributos requisitados em cada tarefa de manipulação direta, pois quanto maior o número de atributos, maior será o processamento exigido na GPU. Em um primeiro teste para verificar esse comportamento, medimos o tempo de processamento de diferentes tarefas de interação em diferentes modelos geométricos. Assim como no teste anterior, estes testes foram conduzidos em um computador AMD Athlon 64 3500+ de 2.2GHz e 2GB RAM, equipado com uma placa gráfica NVIDIA GeForce 8800



(a) (b) (c) (d)

Fig. 6.2: Modelos de teste para a execução de diferentes tarefas de manipulação direta. (a) Nó; (b) Esfera; (c) Cabeça; (d) Quadrilátero com *relief mapping*.

GTX com 768MB. Os modelos utilizados são mostrados na figura 6.2: o modelo de um nó, com 1.694 vértices; o modelo de uma esfera, com 2.307 vértices; o modelo de uma cabeça, com 17.361 vértices; um quadrilátero mapeado por *relief mapping*, com 4 vértices. Os resultados são mostrados na tabela 6.2 e incluem o tempo de processamento necessário para deformar a geometria tanto com relação aos vértices como com relação aos fragmentos. As funções de deformação são funções identidade, utilizadas apenas para habilitar os estágios de modificação de atributos de vértices e fragmentos.[1] Os valores da tabela também incluem o tempo necessário para calcular as bases tangentes (vetor normal e vetores tangentes alinhados segundo as coordenadas de textura), codificar e decodificar os atributos de fragmentos. Em tarefas de posicionamento restrito, esse tempo também inclui o processamento realizado na aplicação para utilizar esses atributos na mudança de posição do cursor 3D. Os resultados só não incluem o tempo necessário para realizar o cálculo de iluminação e de fato renderizar a cena.

Na tabela 6.2, o rótulo "Pos. restrito à superfície (somente posição)" indica a execução de uma tarefa de restrição à superfície sem o cálculo de bases tangentes. O rótulo "Pos. restrito à superfície (posição e orientação)" indica a tarefa de restrição à superfície com o cálculo de posição e bases tangentes (vetor normal e vetores tangentes alinhados segundo as coordenadas de textura). Nesses dois casos, a região de interesse envolve apenas o *pixel* apontado pelo cursor 2D. O rótulo "Posicionamento restrito a vértices" indica a tarefa de posicionamento restrito aos vértices da malha. Essa tarefa utiliza uma simulação de um campo de gravidade em torno de cada vértice renderizado, conforme o procedimento sugerido na seção 3.2.2 do capítulo 3. Nesta tarefa, utilizamos uma região de interesse de 50×50 *pixels*, centralizada em torno do *pixel* apontado pelo cursor 2D. Os rótulos "Seleção de todas as faces (N interseções)" indicam tarefas de seleção que simulam o comportamento do algo-

[1] Exceto para o quadrilátero com *relief mapping*, pois neste caso o estágio de modificação de atributos de fragmentos contém o próprio *shader* da técnica de *relief mapping*.

	Nó	Esfera	Cabeça	*Relief mapping*
Pos. restrito à superfície (somente posição)	1,33	1,71	3,65	0,39
Pos. restrito à superfície (posição e orientação)	1,62	2,06	4,90	0,39
Posicionamento restrito a vértices	13,67	14,06	16,24	-
Seleção de faces (2 interseções)	2,42	3,16	7,10	-
Seleção de faces (4 interseções)	4,72	-	14,08	-
Seleção de faces (6 interseções)	6,97	-	21,06	-
Seleção de faces (8 interseções)	9,38	-	28,02	-

Tab. 6.2: Tempo médio de processamento, em milissegundos, para executar diferentes tarefas de manipulação usando a arquitetura de interação.

ritmo tradicional de *ray picking* para determinar a posição de todos os pontos de interseção entre a geometria e o raio de seleção. Com a arquitetura proposta, esse comportamento é obtido segundo o procedimento iterativo descrito na seção 3.2.1 do capítulo 3. No modelo da esfera, os resultados para o cálculo de 4, 6 e 8 interseções não são exibidos, pois o número máximo de interseções entre um raio e uma esfera é 2. A coluna "*Relief mapping*" indica o tempo obtido em cada tarefa de interação sobre um quadrilátero mapeado com detalhes 3D segundo o método de *relief mapping*. O posicionamento restrito a vértices e a seleção de faces não se aplica nesse caso.

Comparando os resultados das duas primeiras linhas da tabela 6.2 para os modelos do nó, esfera e cabeça, verificamos que o cálculo das bases tangentes aumenta o tempo de processamento de forma proporcional à complexidade dos modelos utilizados. Este custo é reflexo do uso do estágio de cálculo de atributos geométricos (estágio 2 da arquitetura segundo as figuras 5.9 e 5.10), pois tal estágio emprega *shaders* com instruções de fluxo de controle dinâmico e laços contendo instruções de amostragem de textura para ler os dados de atributos de vértices e dados de conectividade. Para o quadrilátero com *relief mapping*, o tempo de processamento é desprezível. Isso ocorre porque, neste modelo, os atributos de posição e orientação são calculados pelo próprio algoritmo de *relief mapping*, o qual foi fornecido pela aplicação para ser executado no estágio de modificação de atributos de fragmentos. Embora o gargalo do algoritmo de *relief mapping* se encontre no processamento de *pixels*, neste caso o processamento se aplica a apenas um *pixel*, que é o *pixel* determinado pela região de interesse. Como esse processamento é realizado no processador de fragmentos, o estágio de cálculo de atributos geométricos não precisa ser ativado.

O tempo de processamento do posicionamento restrito a vértices é significativamente maior do que o tempo de processamento do posicionamento restrito à superfície. Uma vez que os mesmos atributos são calculados nas duas tarefas, concluímos que esse tempo adicional é resultado do aumento do tamanho da região de interesse. Uma região de interesse maior se reflete em uma maior taxa de preenchimento dos *buffers* de visualização não visíveis e uma maior quantidade de dados que devem

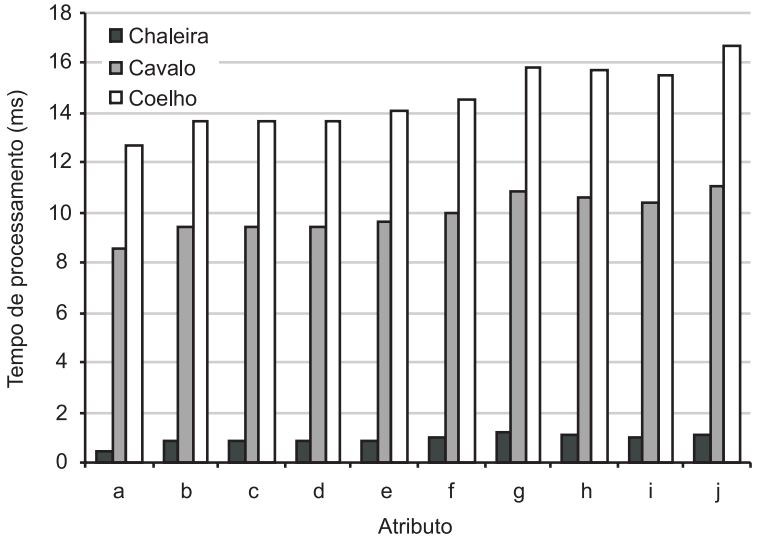


Fig. 6.3: Tempo de processamento, em milissegundos, de diferentes atributos utilizando o fluxo completo de processamento da arquitetura.

ser transferidos da GPU para a CPU durante a etapa de decodificação de atributos.

As medições obtidas da tarefa de selecionar todas as faces intersectadas pelo raio de seleção mostram que cada duas novas interseções adicionam um tempo constante de processamento (em torno de ~3 ms para o modelo do nó e ~7 ms para a o modelo da cabeça). Esse tempo constante é o tempo de processamento adicional da execução dos estágios da arquitetura para cada iteração do procedimento de simulação de *ray picking* descrito na seção 3.2.1. Observe que, para essa tarefa de seleção, todos os estágios da arquitetura são executados, com exceção dos estágios de modificação de atributos de vértices e cálculo de atributos geométricos.

Para verificar isoladamente o impacto do uso de cada atributo no desempenho da arquitetura, mostramos na figura 6.3 o tempo médio, em milissegundos, obtido para calcular diferentes atributos em um modelo com 2.082 vértices (chaleira), 48.484 vértices (cavalo) e 72.027 vértices (coelho). Esse tempo inclui a execução dos estágios de modificação de atributos de vértices, modificação de atributos de fragmentos, codificação e decodificação de atributos. Os atributos calculados são: (a) nenhum atributo; (b) valor de profundidade; (c) coordenadas de textura; (d) valor definido pela aplicação para geometria indexada; (e) valor definido pela aplicação para geometria não indexada; (f) vetor normal; (g) base tangente composta de vetor normal e vetores tangentes alinhados de acordo com as coordenadas de textura; (h) curvaturas principais e direções principais; (i) coeficientes do tensor de curvatura; (j) coeficientes do tensor de derivada de curvatura.

A sobrecarga de processamento mostrada em (a) é a sobrecarga inerente à arquitetura, com peso maior distribuído na execução das instruções de fluxo de controle dinâmico e instruções de

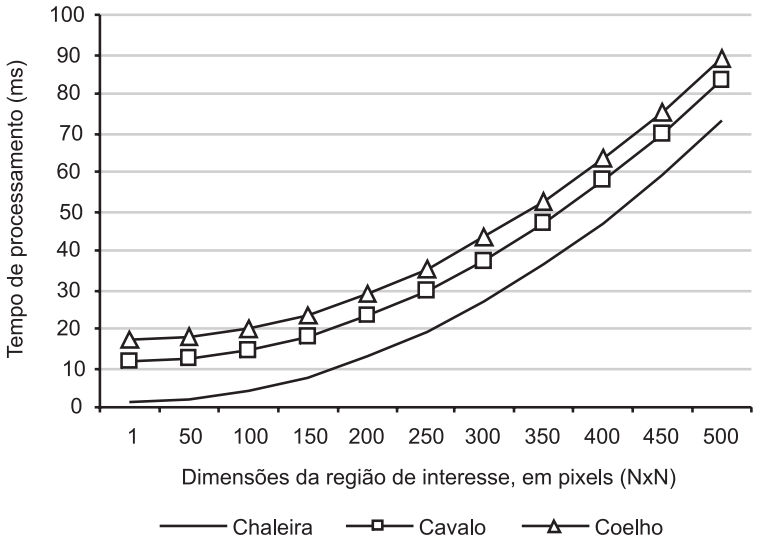


Fig. 6.4: Tempo de processamento, em milissegundos, de um conjunto de atributos em função do tamanho da região de interesse.

amostragem de textura utilizadas no estágio de modificação de atributos de vértices. Levando em consideração tal sobrecarga, observamos que o processamento dos atributos de valor de profundidade, coordenadas de textura e valores definidos pela aplicação para geometria indexada aumentam apenas de forma sutil o tempo total de processamento (em torno de 7% para o modelo do coelho). O uso de atributos definidos pela aplicação para geometria não indexada é ligeiramente menos eficiente do que o uso em geometria indexada, uma vez que o tamanho do *buffer* de vértices da geometria é maior na versão não indexada do modelo.

Os atributos de maior custo computacional são os atributos de geometria diferencial. Isto ocorre porque nesses casos o estágio de cálculo de propriedades geométricas é utilizado. Como é esperado, o cálculo das direções principais e curvaturas principais é mais custoso do que o cálculo dos coeficientes do tensor de curvatura, uma vez que as direções e curvaturas principais requerem o cálculo do tensor de curvatura para extração dos autovetores e autovalores. Da mesma forma, o cálculo dos coeficientes do tensor de derivada de curvatura é o mais custoso de todos, pois requer o cálculo das direções e curvaturas principais. Ainda assim, convém ressaltar que esse custo de estimativa de elementos de geometria diferencial é muito inferior ao custo de uma estimativa equivalente baseada na CPU, conforme mostramos na seção 4.3.

Para medir a variação do desempenho da arquitetura em função do tamanho da região de interesse, medimos o tempo de execução do cálculo de um conjunto de atributos (valor de profundidade, valor definido pela aplicação para geometria indexada, base tangente, coeficientes do tensor de curvatura) para os três modelos utilizados no teste anterior, e utilizando todos os estágios do fluxo de processa-

mento da arquitetura. O tempo medido inclui a transferência dos dados da região de interesse para a CPU. Os resultados são mostrados na figura 6.4 e mostram que o tamanho da região de interesse contribui de forma linear para a redução do desempenho da arquitetura (note que o eixo das abscissas corresponde à raiz quadrada do número de *pixels*). Isto ocorre porque, internamente, o *shader* de fragmentos que encapsula o estágio de modificação de atributos de fragmentos e o estágio de codificação desses atributos não realiza um processamento intensivo (a função de modificação de atributos de fragmentos é uma função-identidade). Podemos, portanto, concluir que o gargalo do fluxo de processamento está de fato no estágio de cálculo de atributos geométricos.

6.2 Exemplos de aplicações

A seguir, descrevemos como utilizamos a arquitetura de interação para implementar algumas ferramentas de manipulação direta em geometria deformada na GPU.

De modo a validar nossa hipótese de que atributos geométricos e atributos definidos pela aplicação codificados para cada *pixel* do modelo geométrico renderizado são suficientes para suportar a implementação de diferentes tarefas de manipulação direta baseadas em tarefas de seleção 3D e posicionamento restrito 3D com o uso de um dispositivo apontador 2D, mostramos a seguir exemplos de implementação de tarefas de interação 3D utilizando os comandos da biblioteca apresentada no apêndice C. Essas tarefas são: seleção do modelo visível, identificação das faces intersectadas pelo raio de seleção, seleção usando mapas de interação [Pierce and Pausch, 2003], posicionamento restrito a superfícies, posicionamento restrito a vértices, posicionamento restrito a bordas, pintura e escultura de um modelo com *relief mapping*, e posicionamento restrito de acordo com as curvaturas e direções principais. A seguir descrevemos cada uma dessas técnicas, destacando como elas se diferem em relação aos comandos da biblioteca responsáveis pela interface de entrada da arquitetura. Em todas as tarefas de interação, a ordem de execução de outros comandos da biblioteca segue a ordem sugerida na seção 5.5.

6.2.1 Seleção do modelo visível

A tarefa de seleção do modelo visível consiste em identificar qual modelo geométrico, dentre todos os modelos visíveis na tela, está sendo apontado pelo cursor 2D de posicionamento livre. Uma realimentação visual comum para seleção consiste em destacar o modelo apontado pelo cursor 2D. Isto pode ser feito, por exemplo, alterando a cor do modelo selecionado. Uma imagem da aplicação utilizando esta tarefa é mostrada na figura 6.5. Nesta figura, o modelo selecionado pelo cursor 2D é destacado com outra cor, e o valor do identificador é mostrado na parte inferior da imagem.

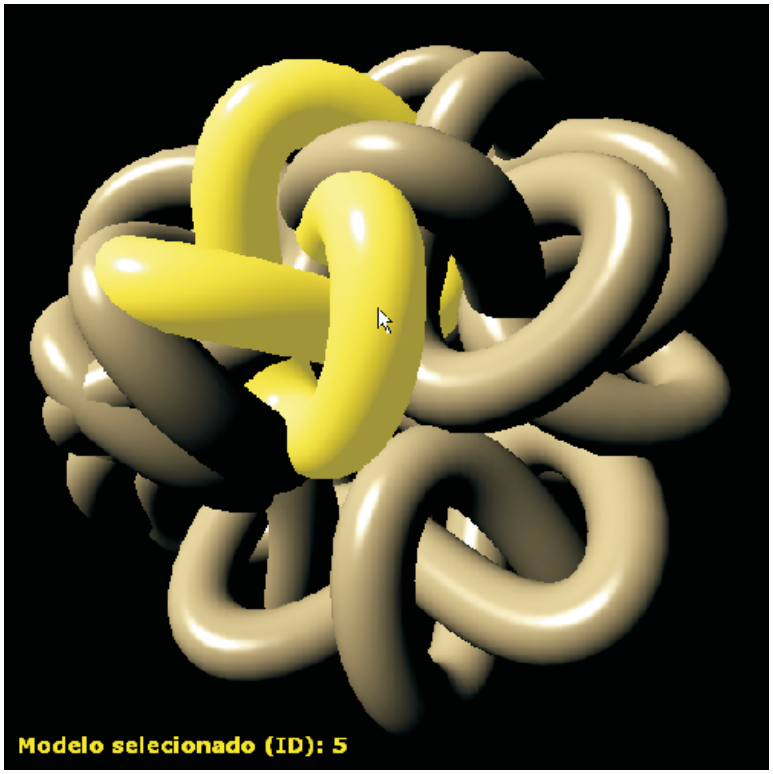


Fig. 6.5: Seleção do modelo visível.

Os seguintes comandos da biblioteca são utilizados para inicializar o fluxo de processamento da arquitetura para a tarefa de seleção e obter os atributos calculados:

- Chamamos o comando `CIntManager::SetAttributes()` passando como parâmetro apenas o valor de enumeração `ATTTYPE_USERDEFI`. Isto é utilizado para informar que o único atributo a ser armazenado para cada *pixel* do modelo renderizado é um valor definido pela aplicação para a geometria indexada do modelo. Neste caso, tal valor é o identificador do modelo, incluído no *buffer* de vértices da geometria original como um elemento adicional de coordenadas de textura.

- Chamamos o comando `CIntManager::BindSemantics()`, utilizando como parâmetros o par de valores `INTSEMANTIC_TEXCOORD0` e `INTSEMANTIC_VERTEXID` de modo a informar que o identificador de cada vértice é encontrado no primeiro atributo de coordenadas de textura do *buffer* de vértices,[2] e o par de valores `VSSEMANTIC_TEXCOORD1` e `INTSEMANTIC_USERDEFI` de modo a informar que o valor definido pela aplicação para

[2]Para todos os exemplos de aplicações nesta seção, assumimos este mapeamento semântico entre o identificador de vértices e o primeiro atributo de coordenadas de textura.

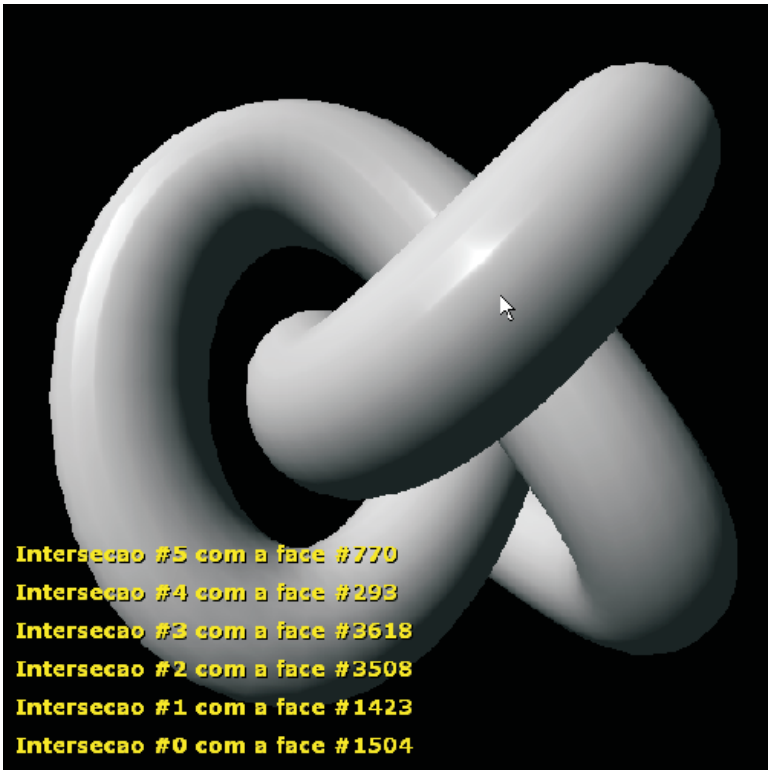


Fig. 6.6: Seleção de todas as faces intersectadas pelo raio de seleção.

a geometria indexada é encontrado no segundo atributo de coordenadas de textura.

- Utilizamos o comando `CIntManager::SetROI()`, passando como coordenadas a localização atual do cursor 2D. Este comando é chamado sempre que um novo evento de movimentação do cursor é disparado pelo sistema de janelas.

- No fim de cada iteração do laço de renderização, o comando `CIntManager::Decode()` é utilizado para obter o identificador armazenado no *pixel* apontado pelo cursor 2D. Esse identificador é utilizado pela aplicação para realizar a realimentação visual.

6.2.2 Seleção de faces intersectadas pelo raio de seleção

Esta tarefa consiste em determinar os identificadores de todas as faces que fariam interseção com o raio de seleção, caso o método tradicional de *ray picking* fosse utilizado. O procedimento necessário para essa tarefa é o procedimento iterativo de seleção descrito na seção 3.2.1, adaptado para faces: (1) seleciona-se a face visível; (2) renderiza-se a cena novamente, porém excluindo a(s) face(s) já selecionada(s); (3) repete-se o procedimento a partir do primeiro passo até que o *pixel* apontado

pelo cursor 2D não contenha mais nenhum identificador de face. Uma vez que cada face deve ter seu identificador próprio, utilizamos geometria não indexada, pois os identificadores das faces são definidos pelos atributos dos vértices que compõem cada face.

A captura de uma imagem desta aplicação é mostrada na figura 6.6. Os identificadores de cada face intersectada, em uma ordem de frente para trás a partir do observador, são exibidos no canto inferior esquerdo da imagem.

Os seguintes comandos da biblioteca são utilizados para configurar o fluxo de processamento da arquitetura e ler os resultados:

- Utilizamos o comando `CIntManager::SetAttributes()` passando como parâmetro apenas o valor de enumeração `ATTTYPE_USERDEFNI`. Este parâmetro informa que o único atributo a ser armazenado para cada *pixel* do modelo renderizado é um valor definido pela aplicação para a geometria não indexada do modelo. Tal valor é o identificador de cada face, e é incluído no *buffer* de vértices da geometria original como um elemento adicional de coordenadas de textura.

- O comando `CIntManager::BindSemantics()` é chamado com o par de valores `VSSEMANTIC_TEXCOORD1` e `INTSEMANTIC_USERDEFNI` de modo a informar que o valor definido pela aplicação para a geometria não indexada é encontrado no segundo atributo de coordenadas de textura do *buffer* de vértices.

- O comando `CIntManager::SetROI()` é utilizado da mesma forma que na tarefa de seleção, passando como coordenadas a localização atual do cursor 2D.

- Chamamos o comando `CIntObj::SetPixelDeform()` do modelo sob interação para definir um *shader* de modificação de atributos de fragmentos que descarta a renderização dos fragmentos de todas as faces selecionadas anteriormente no processo iterativo de seleção de faces. A informação sobre quais faces já foram selecionadas é obtida de registradores constantes configurados pela aplicação nas chamadas de retorno do estágio de codificação de atributos.

- No fim de cada iteração do fluxo de processamento da arquitetura, o comando `CIntManager::Decode()` é utilizado para obter o identificador da face visível na atual iteração. Nas funções de chamada de retorno utilizadas no estágio de codificação de atributos (*i.e.*, definidas com `CIntObj::SetRenderCallback()`), os identificadores das faces já selecionadas são enviados ao *shader* de modificação de atributos de fragmentos, o qual utilizará esses valores para determinar quais as faces que não devem ser renderizadas.

Fig. 6.7: Seleção usando mapas de interação.

6.2.3 Seleção usando mapas de interação

Através de mapeamento de textura é possível aplicar um arranjo retangular de dados sobre a geometria, de acordo com uma parametrização definida pelas coordenadas de textura. De acordo com Pierce and Pausch [2003], quando esses dados mapeados sobre o modelo são utilizados para disparar eventos sempre que um cursor 2D com posicionamento livre aponte os *pixels* contendo tais valores mapeados, estes mapas de textura são chamados de *mapas de interação*. Uma vez que os *texels* da textura podem conter diferentes valores de disparo de eventos, o uso desses mapas é particularmente útil para realizar interações acuradas em modelos baseados em imagem e modelos cujas texturas contém a maior parte dos detalhes da geometria.

No exemplo de aplicação de mapas de interação usando nossa arquitetura, aplicamos um mapa de interação sobre um modelo geométrico e, para cada evento de seleção de um valor do mapa (que neste caso é um valor de cor), a realimentação visual consiste em alterar a cor do modelo de acordo com a cor apontada pelo cursor 2D. Uma imagem desta aplicação é mostrada na figura 6.7. Nesta figura, os valores de disparo de eventos do mapa de interação são três, e correspondem às cores amarela, azul e vermelha mapeadas sobre o modelo. A cor ambiente do modelo é modificada de acordo com o valor

de cor obtido do mapa de interação. Nesta imagem, o modelo é mostrado com a cor ambiente azul pois o cursor 2D está apontando uma região azul do mapeamento do mapa de interação. Os seguintes comandos da biblioteca são utilizados para inicializar o fluxo de processamento da arquitetura e obter os atributos calculados:

- Assim como na tarefa de seleção, chamamos o comando `CIntManager::SetAttributes()` passando como parâmetro apenas o valor de enumeração `ATTTYPE_USERDEFI`. Tal valor é o valor de cada *texel* do mapa de interação, amostrado no estágio de modificação de atributos de fragmentos. O *buffer* de vértices da geometria original não precisa ser modificado. Assim, o comando `CIntManager::BindSemantics()` não precisa ser chamado para informar atributos além do identificador do vértice.
- Utilizamos o comando `CIntObj::SetPixelDeform()` para definir um *shader* de fragmentos que amostra o mapa de interação segundo as coordenadas de textura da geometria original, e armazena o valor amostrado no registrador de saída utilizado para armazenar o valor definido pela aplicação.
- Assim como na tarefa de seleção, a região de interesse é o *pixel* apontado pelo cursor 2D. Isto é configurado com o comando `CIntManager::SetROI()`.
- No final de cada iteração do laço de renderização, o comando `CIntManager::Decode()` é utilizado para obter o identificador armazenado no *pixel* apontado pelo cursor 2D. Esse identificador é utilizado pela aplicação para realizar a realimentação visual.

6.2.4 Posicionamento restrito a superfícies

Em nossa aplicação de posicionamento restrito a superfícies, deslocamos um cursor tríade para o ponto da superfície apontado pelo cursor 2D e orientamos seus eixos de acordo com os atributos de propriedades de geometria diferencial estimados para aquele ponto da superfície. Em particular, alinhamos o cursor tríade tal forma que um de seus eixos aponta na direção do vetor normal (seta vermelha) e os outros dois eixos perpendiculares apontam nas direções dos vetores tangente e bitangente (setas verde e azul, respectivamente). A simulação de um campo de atração não é utilizada neste caso. Uma imagem desta aplicação é mostrada na figura 6.8. Os valores de posição e orientação são mostrados no texto no canto inferior esquerdo da imagem.

Os seguintes comandos da biblioteca são utilizados para inicializar o fluxo de processamento da arquitetura e obter os atributos necessários para posicionar e orientar o cursor tríade:

- Chamamos o comando `CIntManager::SetAttributes()`, passando como parâmetros os valores de enumeração `ATTTYPE_DEPTH` e `ATTTYPE_TBN`. Isso significa que os atributos

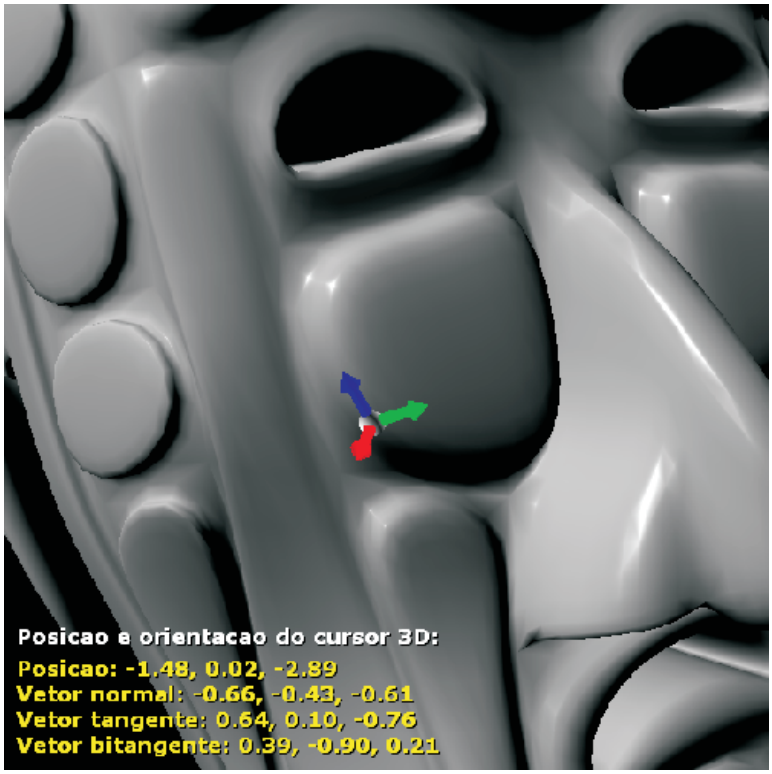


Fig. 6.8: Posicionamento restrito a superfícies.

calculados serão os valores de profundidade e bases tangentes alinhadas de acordo com a parametrização das coordenadas de textura. O valor de profundidade é utilizado para determinar a posição 3D do ponto de restrição, e as bases tangentes são utilizadas para orientar o cursor tríade sobre o plano tangente à superfície no ponto. Ao contrário das tarefas que utilizam valores definidos pela aplicação, nesta tarefa de posicionamento restrito a superfícies o *buffer* de vértices da geometria original não precisa ser alterado para inclusão de novos atributos.

- Utilizamos o comando `CIntManager::BindSemantics()` com o par de valores `VSSEMANTIC_TEXCOORD0` e `INTSEMANTIC_TEXCOORD` de modo a informar que as coordenadas de textura utilizadas para o cálculo dos vetores tangente e bitangente são encontradas no primeiro atributo de coordenadas de textura do *buffer* de vértices.

- O comando `CIntManager::SetROI()` é utilizado da mesma forma que na tarefa de seleção, passando como coordenadas a localização atual do cursor 2D.

- No final de cada iteração do laço de renderização, o comando `CIntManager::Decode()` é utilizado para obter o valor de profundidade no *pixel* apontado pelo cursor 2D, e as estimativas

Fig. 6.9: Posicionamento restrito a vértices.

do vetor normal e vetores tangentes. Esses dados são utilizados pela aplicação para posicionar o cursor tríade na superfície e orientá-lo de acordo com a base tangente.

6.2.5 Posicionamento restrito a vértices

Esta tarefa é semelhante ao posicionamento restrito a superfícies, pois também envolve o posicionamento de um cursor 3D em pontos da superfície visualizada. Entretanto, neste caso o cursor só é posicionado nos vértices da malha. Para obter este comportamento, simulamos a influência de um campo de gravidade em torno de cada vértice. Quando o cursor 2D se aproxima deste campo, é automaticamente deslocado para a posição do vértice. Além disso, não calculamos a orientação do cursor no ponto de restrição. Uma imagem desta aplicação é mostrada na figura 6.9. O cursor 3D é exibido como uma pequena esfera vermelha. Para facilitar a compreensão, mostramos o conteúdo da região de interesse centralizada na posição do cursor. Os pontos brancos da região de interesse correspondem aos *pixels* que possuem atributos codificados. Para uma melhor visualização, o tamanho de cada ponto foi aumentado nesta imagem (na realidade, cada ponto corresponde a apenas um *pixel*). Esses *pixels* são os *pixels* coincidentes com a rasterização dos vértices da malha.

Nesta tarefa de interação, utilizamos os seguintes comandos da biblioteca para inicializar o fluxo de processamento da arquitetura e obter os atributos calculados:

- Utilizamos o comando `CIntManager::SetAttributes()`, passando como parâmetro apenas o valor de enumeração `ATTTYPE_DEPTH`. Isto significa que apenas o valor de profundidade será calculado. Esse valor é utilizado pela aplicação para determinar a posição 3D do cursor restrito. O comando `CIntManager::BindSemantics()` não é utilizado neste caso para informar a ligação semântica de atributos além do identificador dos vértices.

- Com o comando `CIntManager::SetROI()`, definimos uma região retangular em torno da posição atual do cursor 2D. O tamanho dessa região define o tamanho da área de atuação do campo de gravidade.

- Na função de chamada de retorno especificada em `CIntObj::SetRenderCallback()`, o estado de renderização da API gráfica é alterado de modo a renderizar apenas os vértices da geometria.

- No final de cada iteração do laço de renderização, o comando `CIntManager::Decode()` é utilizado para obter os valores de profundidade de todos os *pixels* da região de interesse. Uma vez que só os vértices da geometria foram renderizados, apenas os *pixels* coincidentes com esses vértices terão atributos calculados. A aplicação determina qual *pixel* com atributo calculado se encontra mais próximo da atual posição do cursor 2D. O atributo de valor de profundidade deste *pixel* mais próximo é utilizado para determinar a posição 3D do cursor 3D restrito.

6.2.6 Posicionamento restrito a bordas

Assim como na tarefa anterior restringimos o posicionamento do cursor 3D a vértices da malha renderizada, nesta tarefa restringimos o posicionamento do cursor 3D a pontos da superfície coincidentes com bordas da imagem. Em nossa aplicação, essas bordas foram detectadas com o filtro de detecção de bordas de Roberts [1963].

Esta tarefa ilustra como a arquitetura proposta pode ser utilizada para realizar interações com modelos cuja visualização é afetada por filtros de pós-processamento de imagens. Uma imagem desta aplicação é mostrada na figura 6.10. De forma semelhante à tarefa anterior, o cursor 3D é exibido como uma pequena esfera vermelha, e o conteúdo da região de interesse é exibido. Os pontos brancos da região de interesse correspondem aos *pixels* que possuem atributos codificados, *i.e.*, os *pixels* coincidentes com as bordas detectadas.

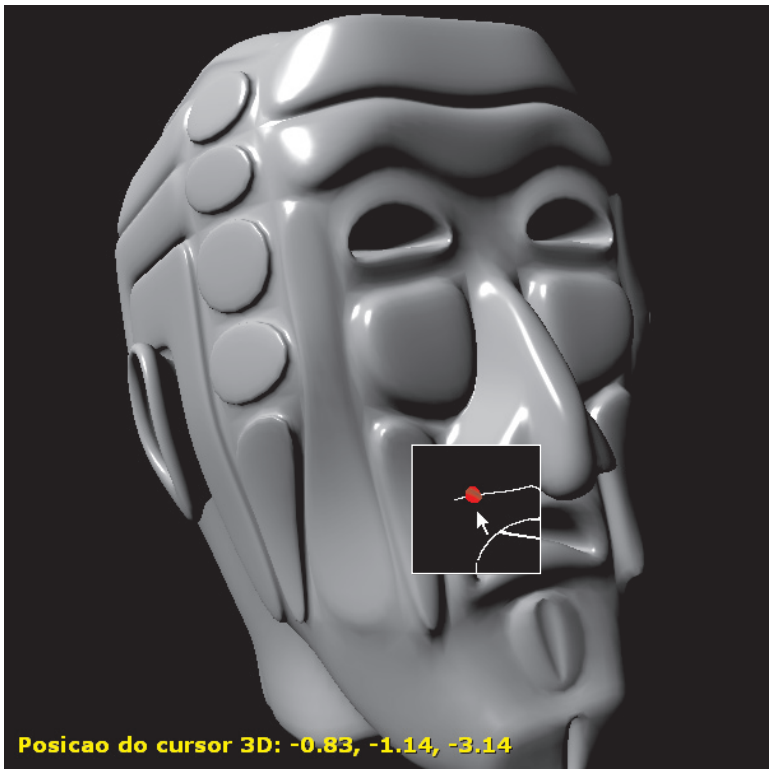


Fig. 6.10: Posicionamento restrito a bordas.

Os seguintes comandos da biblioteca foram utilizados para inicializar o fluxo de processamento da arquitetura e obter os atributos calculados:

- Utilizamos o comando `CIntManager::SetAttributes()`, passando como parâmetro apenas o valor de enumeração `ATTTYPE_DEPTH`. Isto significa que apenas o valor de profundidade será calculado. Esse valor é utilizado pela aplicação para determinar a posição 3D do cursor restrito. O comando `CIntManager::BindSemantics()` não é utilizado neste caso para informar a ligação semântica de atributos além do identificador dos vértices.

- Com o comando `CIntManager::SetROI()`, definimos uma região retangular em torno da posição atual do cursor 2D. O tamanho dessa região define o tamanho da área de atuação do campo de gravidade.

- No final de cada iteração do laço de renderização, o comando `CIntManager::Decode()` é utilizado para obter os valores de profundidade de todos os *pixels* da região de interesse. Neste caso, as bordas ainda não foram detectadas. Para isolar os *pixels* coincidentes com as bordas, a aplicação precisa executar o filtro de detecção de bordas sobre a imagem final e então utilizar

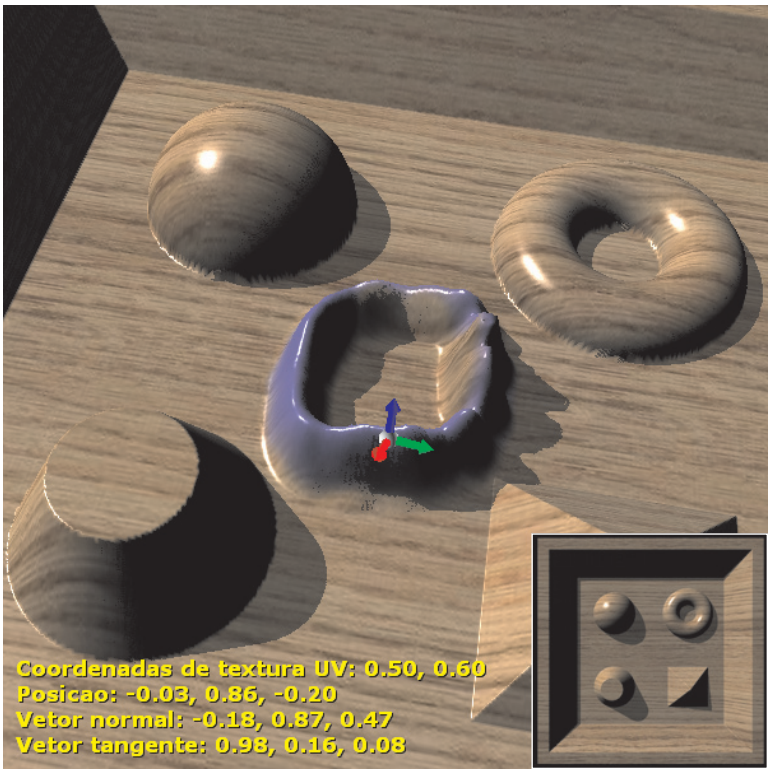


Fig. 6.11: Pintura e escultura de um quadrilátero mapeado com *relief mapping*.

a imagem resultante como uma máscara sobre os valores obtidos da região de interesse. Desse modo, a própria aplicação determina quais atributos serão ignorados. O procedimento restante para realimentação visual pode ser feito como na tarefa de posicionamento restrito a vértices.

6.2.7 Pintura e escultura de um modelo com *relief mapping*

Uma aplicação simples de pintura 3D pode ser implementada através da leitura das coordenadas de textura armazenadas na região de interesse. Nesta aplicação, as coordenadas de textura podem ser utilizados para modificar o conteúdo dos mapas de textura correspondentes. Com a arquitetura proposta, tais coordenadas podem ser disponibilizados para cada *pixel* da imagem renderizada e podem levar em consideração possíveis modificações desses atributos em *shaders* de fragmentos. Desse modo, podemos implementar uma aplicação de pintura 3D que trabalhe com modelos renderizados com técnicas de mapeamento de detalhes 3D. Mesmo que tais técnicas modifiquem os atributos de coordenadas de textura, profundidade e vetor normal de cada *pixel*, a tarefa de interação será consistente com aquilo que o usuário estará visualizando.

Para mostrar o potencial da arquitetura em casos que envolvem a modificação de atributos de

fragmentos no *shader* de fragmentos, implementamos uma aplicação de pintura e escultura de um quadrilátero mapeado com *relief mapping* [Policarpo et al., 2005]. Uma imagem dessa aplicação é mostrada na figura 6.11. A aparência do quadrilátero visualizado antes da pintura e escultura é mostrado no canto inferior direito. Para uma melhor realimentação visual, um cursor tríade é posicionado de forma restrita à superfície visualizada, e sua oclusão com relação aos detalhes exibidos é considerada corretamente (*e.g.*, parte do cursor tríade pode ficar escondida por detalhes da textura, e vice-versa). Uma ferramenta para editar os parâmetros de altura do mapa de altura associado aos detalhes também foi implementada, fornecendo assim ao usuário um meio de esculpir a mesoestrutura usando manipulação direta.

Os seguintes comandos da biblioteca foram utilizados para inicializar o fluxo de processamento da arquitetura e obter os atributos calculados:

- Chamamos o comando `CIntManager::SetAttributes()`, passando como parâmetros os valores de enumeração `ATTTYPE_DEPTH`, `ATTTYPE_TEXCOORD` e `ATTTYPE_NORMAL`. Isto significa que apenas o valor de profundidade, coordenadas de textura e vetor normal serão calculados.

- Chamamos o comando `CIntManager::BindSemantics()` com o par de valores `VSSEMANTIC_TEXCOORD0` e `INTSEMANTIC_TEXCOORD` para informar que as coordenadas de textura são encontradas no primeiro atributo de coordenadas de textura do *buffer* de vértices.

- Chamamos o comando `CIntObj::SetPixelDeform()` para passar o *shader* de *relief mapping*. Esse *shader* é semelhante ao *shader* utilizado na visualização do modelo. Entretanto, em vez de retornar um valor de cor e um valor de profundidade, o *shader* retorna os atributos modificados de coordenadas de textura, profundidade e vetor normal.

- Utilizamos o comando `CIntManager::SetROI()`, passando como coordenadas a localização atual do cursor 2D.

- No final de cada iteração do laço de renderização, o comando `CIntManager::Decode()` é utilizado para obter as coordenadas de textura, vetor normal e valor de profundidade do *pixel* da região de interesse. Com esses valores, a aplicação pode modificar o conteúdo do mapa de detalhes e realizar o posicionamento e alinhamento do cursor tríade da mesma forma como no posicionamento restrito a superfícies.

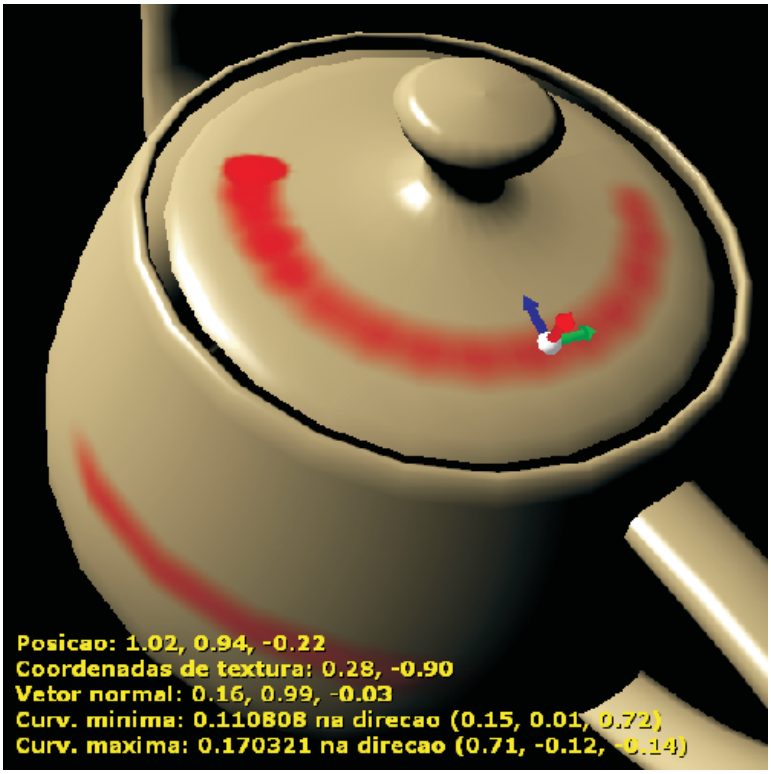


Fig. 6.12: Posicionamento restrito de acordo com as direções principais.

6.2.8 Posicionamento restrito de acordo com as curvaturas e direções principais

Para realizarmos tarefas de posicionamento restrito de acordo com as curvaturas e direções principais, baseamo-nos na idéia de *geometric snapping* proposta por Yoo and Ha [2004] e inspirada na técnica de *image snapping* [Gleicher, 1995]. Nesta técnica, o cursor restrito a um vértice da superfície pode ser deslocado a outro vértice de acordo com a avaliação de uma função de custo de movimento baseada nos atributos de curvatura da superfície. Essa função utiliza os mínimos ou máximos locais das curvaturas principais em cada vértice do modelo de modo a realizar um posicionamento restrito do cursor a vales e topos da superfície.

Na arquitetura proposta, utilizamos o cálculo de atributos geométricos para calcular curvaturas e direções principais de modo a realizar dois tipos de posicionamentos restritos. O primeiro consiste em deslocar um cursor tríade nas direções principais da superfície. Esse tipo de restrição pode ser útil em aplicativos que utilizam as direções principais para realizar *hatching* coerente com a forma do objeto [Elber, 1999, Praun et al., 2001], pois fornece um meio de interagir com os traços através de pintura 3D. Uma ilustração de aplicação simples utilizando este tipo de restrição é mostrada na

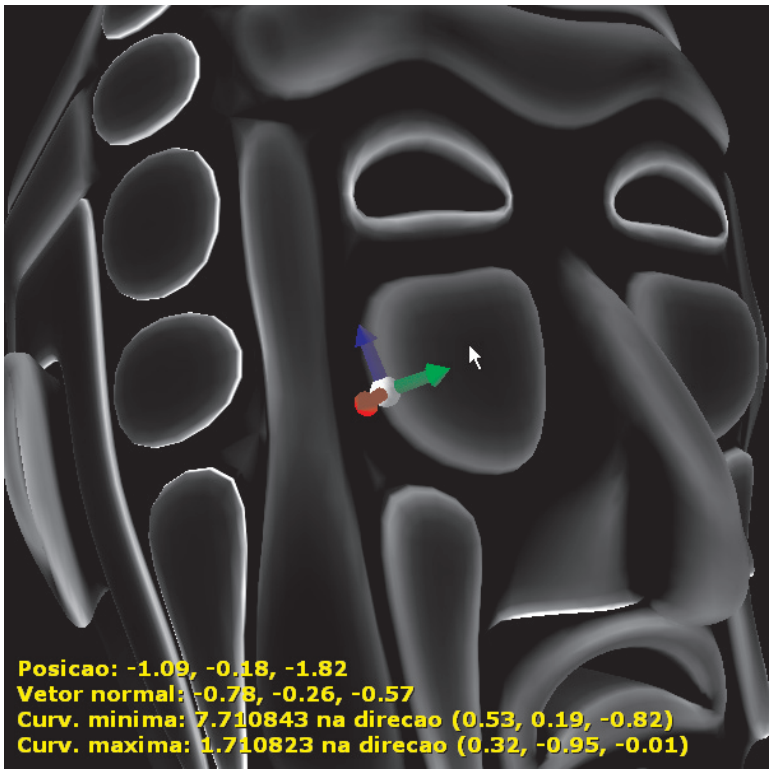


Fig. 6.13: Posicionamento restrito de acordo com a curvatura média.

figura 6.12. Nesta figura, os traços exibidos em vermelho foram desenhado durante a restrição de deslocamento do cursor na direção da curvatura mínima a partir de um determinado ponto sobre a superfície. Em outro tipo de posicionamento restrito, deslocamos suavemente o cursor tríade para o ponto de maior curvatura média em torno da atual posição do cursor 2D. Dessa forma, o cursor tríade tende a se aproximar das regiões com maior curvatura. Uma ilustração dessa aplicação é mostrada na figura 6.13. Nesta figura, o sombreamento do modelo em tons de cinza indica o valor da curvatura média em cada ponto. Cores mais claras indicam valores positivos, e cores mais escuras indicam valores negativos. A cor cinza indica uma curvatura média igual a zero. Nas duas aplicações, o cursor tríade é alinhado de acordo com as direções principais no ponto de restrição.

Os seguintes comandos da biblioteca foram utilizados para inicializar o fluxo de processamento da arquitetura e obter os atributos calculados para as tarefas citadas:

- Chamamos o comando `CIntManager::SetAttributes()`, passando como parâmetros os valores de enumeração `ATTTYPE_DEPTH`, `ATTTYPE_NORMAL` e `ATTTYPE_CURV`. Com isso, os atributos calculados serão o valor de profundidade, vetor normal, curvaturas principais e direções principais. O comando `CIntManager::BindSemantics()` não é utilizado neste caso.

- Para a restrição de deslocamento na direção de curvatura mínima, utilizamos o comando `CIntManager::SetROI()`, passando como coordenadas a localização atual do cursor 2D. Para a restrição de posicionamento de acordo com a maior curvatura média, as coordenadas da região de interesse são as coordenadas de um retângulo centralizado na localização atual do cursor 2D, e que define a área de atuação do campo de gravidade.

- No final de cada iteração do laço de renderização, o comando `CIntManager::Decode()` é utilizado para obter o vetor normal, as curvaturas principais e as direções principais armazenadas nos *pixels* da região de interesse. Com esses valores a aplicação realiza o posicionamento e alinhamento do cursor tríade conforme exige cada tarefa de interação. Por exemplo, na restrição de deslocamento na direção de curvatura mínima, cada evento de movimentação do cursor 2D é interpretado pela aplicação como um deslocamento do cursor tríade na direção principal dessa curvatura. Na restrição de maior curvatura média, a aplicação determina qual o *pixel* da região de interesse que contém a maior curvatura média, e então desloca o cursor tríade até a posição 3D indicada nesse *pixel*.

6.3 Considerações finais

Através da arquitetura proposta, conseguimos implementar diferentes tipos de tarefas de interação 3D baseadas nas tarefas básicas de seleção e posicionamento com restrições. As tarefas implementadas utilizam um fluxo comum de processamento para obter os atributos geométricos e não geométricos dos pontos de interesse. Cabe à aplicação apenas a tarefa de configurar a interface de entrada e saída desse fluxo de modo a determinar quais atributos deverão ser calculados e em qual região da tela eles deverão ser disponibilizados. Como o processamento é integrado ao fluxo de visualização das GPUs, o cálculo de atributos geométricos pode levar em consideração as modificações dos atributos originais da geometria, e é mais eficiente do que o processamento análogo utilizando o método tradicional de *ray picking*.

Tarefas complexas de manipulação direta, tais como *geometric snapping* [Yoo and Ha, 2004], posicionamento do cursor com simulação de retorno de força (*force-feedback*) [van Mensvoort, 2002], pintura 3D utilizando *hatching* [Praun et al., 2001] ou detecção de contornos sugestivos [DeCarlo et al., 2003] não foram implementadas porque elas dependem de algoritmos específicos na CPU, ainda em pesquisa, e que fogem do escopo deste trabalho de tese. Entretanto, tais tarefas podem ser implementadas através da composição das tarefas básicas estudadas.

Ao longo do desenvolvimento deste trabalho tivemos a oportunidade de executar os testes de desempenho da arquitetura em diferentes CPUs e GPUs. Os resultados apresentados aqui correspondem apenas ao *hardware* gráfico mais recente ao qual tivemos acesso no desenvolvimento deste trabalho.

Resultados obtidos em GPUs de gerações anteriores, como a GPU NVIDIA GeForce 7900 GTX e NVIDIA GeForce 6200 podem ser encontrados em Batagelo and Wu [2007a] e mostram que o comportamento de desempenho da arquitetura é o mesmo em todas essas gerações, diferindo entre si apenas por um fator constante.

Capítulo 7

Conclusões e trabalhos futuros

Nos últimos anos, as GPUs atingiram níveis de flexibilidade tais que permitem que seu uso hoje seja explorado para realizar tarefas que extrapolem o processamento relacionado à área de síntese de imagens. Atualmente, GPUs têm sido utilizadas para realizar tarefas relacionadas a modelagem geométrica, animação e processamento de imagens. As GPUs também podem ser utilizadas como processadores de fluxo de propósito geral para realizar tarefas tais como ordenação de dados, simulação de dinâmica de corpos rígidos, detecção de colisão e processamento de sinais de áudio [GPGPU, 2007, Luebke et al., 2004b]. Isto é resultado do alto poder computacional que esse tipo de *hardware* conquistou nos últimos anos para o processamento de fluxos de dados, e que supera a evolução do poder computacional das CPUs para esse tipo de processamento. Atualmente, a largura de banda do barramento de acesso aos dados da memória de vídeo local pode ser até um grau de magnitude superior à largura de banda do barramento utilizado nas CPUs. Além disso, o uso de processamento na GPU permite realizar processamento paralelo à CPU.

Contrastando com a utilização das potencialidades das GPUs para realizar tarefas de visualização, modelagem geométrica, animação e tarefas relacionadas a processamento de propósito geral, verificamos que o potencial deste tipo de *hardware* para o suporte à implementação de tarefas de interação 3D ainda não é devidamente explorado, seja para realizar interações que levem em consideração as modificações de atributos de geometria realizadas na GPU, seja para obter interações mais eficientes em comparação com os tradicionais métodos de interação baseados no cálculo de interseções entre um raio de seleção e a geometria (*ray picking*).

Ainda hoje, as mais conhecidas bibliotecas de grafo de cena com suporte a interações de manipulação direta, tais como o OpenSceneGraph [OpenSceneGraph, 2007] e OpenSG [OpenSG, 2007], utilizam a tradicional técnica de *ray picking* para implementar tarefas de seleção e posicionamento com restrições. Isto também é verdade para a maioria dos aplicativos de modelagem geométrica, animação e pintura 3D, tais como o Autodesk AutoCAD [Autodesk, 2007b], Autodesk 3ds Max [Autodesk,

2007a], Autodesk Maya [Autodesk, 2007c], Right Hemisphere Deep Creator/Deep Paint 3D [Hemisphere, 2007], Pixologic ZBrush [Pixologic, 2007], Interactive Effects Amazon Paint [Effects, 2007], e também para a maior parte das bibliotecas *game engines* utilizadas em desenvolvimento de jogos. Esta opção de projeto, baseada no paradigma da Arquitetura Gráfica Unificada [Zeleznik et al., 1991], se mostrou bem sucedida na década de 1990. Hoje, entretanto, apresenta limitações ante o advento de *hardware* gráfico programável. Em particular, tais aplicativos não são capazes de realizar cálculos corretos de interseção com modelos cuja geometria tenha sido deformada na GPU. Por exemplo, se os atributos da geometria original são modificados no processador de vértices ou processador de fragmentos, a geometria visualizada poderá ser muito diversa da geometria original sobre a qual o algoritmo de *ray picking* é aplicado.

A contribuição principal desta tese é a proposta de uma arquitetura de *software* que fornece suporte à implementação de tarefas de manipulação direta 3D usando dispositivos apontadores 2D, integrada à atual arquitetura de *hardware* das GPUs. Ao contrário das abordagens baseadas no tradicional método de *ray picking*, as tarefas de interação baseadas nesta nova arquitetura são capazes de levar em consideração, de forma eficiente, modificações dos atributos das geometrias processadas na GPU.

A arquitetura proposta é baseada em duas hipóteses. A primeira hipótese supõe que os atributos necessários para realizar tarefas básicas de manipulação direta, tais como seleção e posicionamento restrito, se resumem em atributos geométricos e atributos definidos pela aplicação para cada *pixel* do modelo renderizado. Mostramos que tais atributos geométricos são propriedades de geometria diferencial tais como o vetor normal, vetor tangente, vetor bitangente, curvaturas principais e direções principais. Para defender tal hipótese, realizamos um estudo de casos com tarefas básicas de manipulação direta 3D, mostrando como cada tarefa tradicionalmente implementada com *ray picking* pode ser implementada segundo a hipótese defendida. Concluindo que os atributos geométricos essenciais são os atributos de geometria diferencial dos modelos tratados, propomos algoritmos na GPU para a estimativa eficiente e robusta de propriedades de geometria diferencial de modelos discretos. As propriedades calculadas são propriedades relacionadas a derivadas parciais de primeira ordem (vetor normal e vetores tangentes alinhados de acordo com o mapeamento das coordenadas de textura), segunda ordem (coeficientes do tensor de curvatura, curvaturas principais e direções principais) e terceira ordem (coeficientes do tensor de derivada de curvatura). Tais algoritmos se destacam como uma contribuição secundária deste trabalho de tese, e podem ser utilizadas isoladamente em outro contexto além do contexto de interação.

A segunda hipótese supõe que a atual arquitetura de *hardware* gráfico programável fornece flexibilidade suficiente para que os atributos geométricos necessários para suportar tarefas básicas de manipulação direta (em especial, seleção e posicionamento com restrição) possam ser processados

diretamente na GPU e codificados no domínio do espaço da imagem como componentes de cor de cada *pixel*. Esses atributos podem ser decodificados pela aplicação de modo a realizar a tarefa de interação. Uma vez que a GPU processa tais atributos, suas possíveis modificações durante o fluxo de visualização podem ser levadas em consideração. Tal hipótese é defendida pela implementação da arquitetura proposta em *hardware* gráfico atual (utilizando o modelo de *shader* 3.0). A implementação dos algoritmos de estimativa de propriedades de geometria diferencial é utilizada internamente pela arquitetura de modo a estimar tais propriedades após um estágio de modificação de atributos de vértices. Desse modo, a estimativa dessas propriedades pode ser feita de tal forma a levar em consideração a geometria deformada no processador de vértices.

A arquitetura é implementada como uma biblioteca de funções em C++ utilizando a API gráfica Direct3D ou OpenGL. Através desta biblioteca a aplicação tem acesso a comandos que podem ser utilizados para informar à arquitetura os atributos desejados no espaço da imagem e os procedimentos de modificação de atributos de vértices e fragmentos dos modelos tratados, além de comandos para executar o fluxo de processamento da arquitetura e obter os atributos calculados, de forma transparente à aplicação. Utilizando esta biblioteca, conseguimos implementar diversas tarefas de interação 3D baseadas nas tarefas básicas de seleção e posicionamento com restrições. Testes de desempenho mostram que tais tarefas são eficientes se comparadas com o método tradicional de *ray picking* para seleção e posicionamento restrito em modelos visíveis. A diferença de desempenho é tanto mais significativa quanto maior é a complexidade dos modelos tratados, uma vez que todo o processamento da geometria para estimativa de atributos geométricos e codificação desses atributos em *pixels* é feita exclusivamente na GPU. Em nossos testes, esse comportamento foi observado em placas de vídeo de diferentes gerações: NVIDIA GeForce 6200, NVIDIA GeForce 7900 GTX e NVIDIA GeForce 8800 GTX.

Requisitos de generalidade, eficiência, robustez e reusabilidade nortearam a proposta da arquitetura. Acreditamos que, através do uso de sua implementação, mostramos que tais requisitos são preenchidos de forma satisfatória. O requisito de generalidade é preenchido como resultado do uso do próprio fluxo de renderização das atuais GPUs para calcular os atributos necessários às tarefas de interação. Através dessa integração, a arquitetura é capaz de trabalhar com todas as primitivas suportadas pela GPU, e possibilita que a estimativa de atributos geométricos leve em consideração as deformações das primitivas em relação a seus vértices. O requisito de eficiência é satisfeito como reflexo do uso do poder computacional da GPU para executar o fluxo de processamento da arquitetura. Para satisfazer o requisito de robustez, empregamos técnicas inéditas de estimativa de propriedades de geometria diferencial que, quando comparadas com técnicas anteriores, fornecem um bom compromisso entre eficiência e robustez da estimativa para modelos com malhas amostradas irregularmente ou com ruídos. Por fim, o critério de reusabilidade é preenchido através da demonstração da pos-

sibilidade do uso da arquitetura para suportar diferentes tarefas de interação e diferentes modelos geométricos deformados na GPU.

Embora as GPUs atuais tenham um poder de processamento muitas vezes superior ao das CPUs, em outros aspectos a arquitetura de *hardware* gráfico, quando comparada com a arquitetura das CPUs, impõe restrições quanto a quantidade de memória disponibilizada, e quanto ao número e tipo de instruções utilizadas. Essas restrições se refletem em limitações no uso da arquitetura proposta. O foco de nossos trabalhos futuros está na análise de tais problemas e na proposta de alternativas para minimizá-los. Dentre as limitações mais importantes encontradas na utilização da arquitetura proposta em comparação com o paradigma tradicional de interação, destacamos os seguintes pontos:

- **Limitação da memória de vídeo.** Para a execução do estágio de estimativa de propriedades de geometria diferencial, é necessária a existência de texturas que atuem como áreas de memória de propósito geral e que contenham, neste caso, os atributos de geometria e conectividade necessários para a estimativa. Para modelos complexos, tais texturas podem consumir um percentual significativo de memória de vídeo.

  Na implementação da arquitetura segundo a biblioteca apresentada no apêndice C, a definição de cada instância da classe `CIntObj` gera internamente um conjunto de 6 a 12 texturas em formato de ponto flutuante de 32 bits (esse número vai depender da quantidade de tipos de atributos requisitados pela aplicação). Cada *texel* de cada textura contém o atributo de um vértice da geometria, de modo que o tamanho de cada textura é proporcional ao número de vértices da malha. Essas texturas são utilizadas na GPU como áreas de memória para processamento de propósito geral, e podem ser acessadas e modificadas nos estágios de modificação de atributos de vértices e cálculo de propriedades geométricos. Em particular, essas texturas incluem os atributos de posição dos vértices, coordenadas de textura, vetor normal, vetores tangentes e bitangentes, coeficientes do tensor de curvatura, coeficientes do tensor de derivada de curvatura, direções principais e curvaturas principais. Além das texturas de atributos de vértices, 3 texturas adicionais são ainda utilizadas para armazenar as informações de conectividade do modelo. Tais texturas são utilizadas no estágio de cálculo de atributos.

  Além das texturas utilizadas internamente pela arquitetura, a GPU ainda deve compartilhar outros recursos gráficos utilizados na aplicação, tais como *shaders* e *buffers* de vértices. Mesmo quando a estimativa de atributos de geometria diferencial não for utilizada, o estágio de modificação de atributos de vértices precisa armazenar os resultados em texturas que serão utilizadas pelas outras etapas. Dessa forma, como a memória de vídeo disponível é geralmente menor do que a memória disponível no sistema, o número máximo de objetos simultaneamente sob interação será menor do que aquele que seria possível no paradigma convencional baseado na existência dos modelos na memória do sistema.

Acreditamos que tal limitação de memória pode ser amenizada caso a aplicação utilize uma abordagem de determinação de visibilidade para excluir os modelos que não estão contidos do volume de visualização, uma vez que estes modelos não são relevantes para as tarefas de seleção e posicionamento restrito. Além disso, podem ser excluídos também os modelos que estão sendo ocultos por outros modelos, caso as tarefas de interação utilizem apenas a geometria de fato visível. A cena é inicialmente armazenada em um banco de dados na CPU. Em tempo de execução, o algoritmo de determinação de visibilidade determina um conjunto conservador de modelos visíveis, levando em consideração as possíveis deformações que esses modelos poderão sofrer no processador de vértices. Apenas este conjunto de modelos é então submetido à arquitetura de interação. Tal procedimento reduz o consumo de memória de vídeo a apenas estes objetos potencialmente visíveis.

- **Dificuldade de disponibilizar à aplicação atributos de geometria em partes não visíveis dos modelos.** Uma vez que a arquitetura é baseada estritamente na suposição de que o usuário espera interagir com aquilo que ele está vendo, poderão surgir dificuldades para realizar interações que requeiram dados não disponíveis na geometria visível. Um exemplo típico é encontrado na técnica de seleção. No método tradicional de *ray picking*, a técnica de seleção geralmente retorna todos os objetos intersectados pelo raio de seleção, e não apenas o objeto mais próximo, que está de fato visível. A realização de tais tarefas em nossa arquitetura pode ser mais complexa do que sua realização no paradigma tradicional. De fato, a realização do teste entre o raio de seleção e cada primitiva produz naturalmente todas as possíveis interseções. Em nosso caso, tal tarefa requer múltiplos passos de renderização para que, em cada passo, sejam filtrados os fragmentos da geometria do modelo visível. Como observamos nos resultados da tabela 6.2 para a tarefa de seleção de todas as faces intersectadas pelo raio de seleção, a execução desses múltiplos passos pode ser ineficiente, pois praticamente todo o fluxo de processamento da arquitetura é executado em cada iteração.

  Um meio de minimizar essa dificuldade é utilizar, na aplicação, uma abordagem híbrida entre o método tradicional de *ray picking* e a arquitetura proposta. Para cada tarefa de interação que necessitar de todos os pontos de interseção entre o modelo e o raio de seleção, a aplicação pode empregar o método de *ray picking* para tratar os objetos que não sofrem deformações, pois neste caso tal interação tende a ser mais eficiente (tal ganho de eficiência vai depender do poder computacional da GPU e da complexidade dos modelos). Nossa arquitetura é utilizada então apenas para tratar os modelos que de fato sofrem as deformações.

- **Ausência da estimativa de elementos de geometria diferencial após a etapa de modificação de atributos de fragmentos.** Em técnicas de mapeamento de detalhes 3D, os atributos de

posição e orientação do vetor normal de cada fragmento são modificados antes da avaliação da equação do modelo de iluminação. Isso implica a invalidação dos elementos de geometria diferencial de grau superior associados a tais fragmentos. A arquitetura atual não dispõe de uma etapa de estimativa dessas propriedades após a modificação de atributos de fragmentos. Tal estimativa é feita apenas após a etapa de modificação dos atributos dos vértices. Desse modo, assumimos que as técnicas de mapeamento de detalhes 3D utilizam o próprio *shader* de modificação de atributos de fragmentos para atualizar tais propriedades segundo o código específico da técnica de mapeamento de detalhes utilizada. Se essa suposição não for verdadeira, os atributos geométricos poderão se tornar inconsistentes em relação ao modelo visualizado pelo usuário.

Acreditamos que a realização da estimativa de propriedades de geometria diferencial após a modificação de atributos de fragmentos é possível caso esta estimativa seja realizada como um processamento baseado da própria imagem, em um passo de renderização adicional. Como este procedimento é realizado no espaço da imagem, técnicas de processamento de *shape from shading* podem ser empregadas neste caso [Saito and Takahashi, 1990b, Sato et al., 1997]. Uma vez que podemos armazenar, para cada *pixel* da imagem, informações sobre as propriedades de geometria diferencial de primeira ordem, a estimativa dos coeficientes do tensor de curvatura pode ser realizada através da leitura desses atributos nos *pixels* vizinhos ao *pixel* sobre o qual tais coeficientes estão sendo estimados. Em outras palavras, os próprios *pixels* vizinhos podem disponibilizar as informações de vizinhança de 1-anel necessárias à estimativa. Em particular, podemos utilizar as instruções `ddx` e `ddy` no *shader* de fragmentos. Tais instruções retornam a derivada parcial de um dado valor do fragmento com relação às coordenadas x e y do fragmento no espaço da tela. Os valores dados podem ser uma posição 3D, vetor normal ou coordenadas de textura armazenadas nas componentes de cor dos *pixels* do *buffer* de renderização não visível. Por outro lado, a estimativa é incorreta nas bordas do modelo renderizado, pois em tais casos os *pixels* vizinhos podem corresponder a fragmentos de partes diversas do modelo, ou até mesmo a fragmentos de diferentes modelos. Além disso, ela deve ser feita para cada quadro de exibição e não apenas quando o modelo é deformado. Sugerimos, como trabalho futuro, uma análise mais detalhada desta possibilidade de estimativa das propriedades de geometria diferencial a partir das informações presentes unicamente na imagem.

Referências Bibliográficas

3Dfx. *Glide 2.2 Reference Manual*. 3Dfx Interactive, San Jose, CA, USA, May 1997.

ACM. Association for computing machinery. `http://www.acm.org/`, April 2007.

Kurt Akeley. Realityengine graphics. In *Proceedings of SIGGRAPH 1993*, pages 109–116, Anaheim, CA, USA, August 1993.

Kurt Akeley and Patrick Hanrahan. Real-time graphics architecture. `http://www.graphics.stanford.edu/courses/cs448a-01-fall`, March 1999. CS448A Course Notes, Stanford University.

Pierre Alliez, David Cohen-Steiner, Olivier Devillers, Bruno Lévy, and Mathieu Desbrun. Anisotropic polygonal remeshing. *ACM Transactions on Graphics*, 22(3):485–493, 2003. ISSN 0730-0301.

E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 3rd edition, 1999. ISBN 0-89871-447-8 (paperback).

Apple. Apple computer. `http://www.apple.com/`, April 2007.

ATI. Ati technologies. `http://www.ati.com`, April 2007a.

ATI. Comparison of ati graphics processing units. `http://en.wikipedia.org/wiki/Comparison_of_ATI_Graphics_Processing_Units`, April 2007b.

Autodesk. 3ds max. `http://www.autodesk.com/3dsmax`, June 2007a.

Autodesk. Autocad. `http://www.autodesk.com/autocad`, June 2007b.

Autodesk. Maya. `http://www.autodesk.com/maya`, June 2007c.

Harlen Costa Batagelo and Shin-Ting Wu. What you see is what you snap: Snapping to geometry deformed on the gpu. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, pages 81–86, Washington, DC, USA, April 2005. ACM Press. ISBN 1-59593-013-2.

Harlen Costa Batagelo and Shin-Ting Wu. An architecture of 3d interaction for models deformed on the gpu. `http://www.dca.fee.unicamp.br/projects/mtk/batagelo`, 2007a.

Harlen Costa Batagelo and Shin-Ting Wu. Estimating curvatures and their derivatives on meshes of arbitrary topology from sampling directions. *The Visual Computer*, 23(9,11):803–812, September 2007b.

William Baxter, Yuanxin Liu, and Ming C. Lin. A viscous paint model for interactive applications: Research articles. *Comput. Animat. Virtual Worlds*, 15(3-4):433–441, 2004. ISSN 1546-4261.

Eric Allan Bier. Skitters and jacks: interactive 3d positioning tools. In *Proceedings of the 1986 Workshop on Interactive 3D Graphics*, pages 183–196, Chapel Hill, NC, USA, 1987. ACM Press. ISBN 0-89791-228-4.

Eric Allan Bier. Snap-dragging in three dimensions. In *SI3D '90: Proceedings of the 1990 Symposium on Interactive 3D Graphics*, pages 193–204, New York, NY, USA, 1990. ACM Press. ISBN 0-89791-351-5.

James F. Blinn. Models of lights reflection for computer synthesized pictures. In *Proceedings of SIGGRAPH 1977*, pages 192–198, 1977.

James F. Blinn. Simulation of wrinkled surfaces. In *SIGGRAPH '78: Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, pages 286–292, New York, NY, USA, 1978. ACM Press.

David Blythe. The direct3d 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006. ISSN 0730-0301.

Phong Bui-Tong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6): 311–317, 1975.

Dean Calver. Accessing and modifying topology on the gpu. In Wolfgang Engel, editor, *ShaderX3: Advanced Rendering with DirectX and OpenGL*. Charles River Media, November 2004.

Manfredo Perdigão Do Carmo. *Differential geometry of curves and surfaces*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1976. ISBN 0-13-212589-7.

Nathan A. Carr and John C. Hart. Painting detail. In *SIGGRAPH '04: Proceedings of the 31st International Conference on Computer Graphics and Interactive Techniques*, pages 845–852, New York, NY, USA, 2004. ACM Press.

F. Cazals and M. Pouget. Estimating differential quantities using polynomial fitting of osculating jets. In *SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, pages 177–187, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. ISBN 1-58113-687-0.

Xin Chen and Francis Schmitt. Intrinsic surface properties from surface triangulation. In *ECCV '92: Proceedings of the Second European Conference on Computer Vision*, pages 739–743, London, UK, 1992. Springer-Verlag. ISBN 3-540-55426-2.

David Cohen-Steiner and Jean-Marie Morvan. Restricted delaunay triangulations and normal cycle. In *SCG '03: Proceedings of the Nineteenth Annual Symposium on Computational Geometry*, pages 312–321, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-663-3.

Graphics Standards Planning Committee. Status report of the graphics standards planning committee of acm siggraph. *Computer Graphics*, 11(3), 1977.

Robert L. Cook. Shade trees. In *Proceedings of SIGGRAPH 1984*, pages 223–231, New York, NY, USA, 1984. ACM Press. ISBN 0-89791-138-5.

Robert L. Cook and Kenneth E. Torrance. A reflectance model for computer graphics. *ACM Transaction on Graphics*, 1(2):7–24, 1982.

Albert J. Corda. This old sgi. `http://www.geocities.com/SiliconValley/Pines/2258/4dfaq.html`, 1996.

Matthias Kalle Dalheimer. *Programming with Qt: Writing Portable GUI applications on Unix and Win32*. O'Reilly, January 2002. ISBN 0-596-00064-2.

William J. Dally. Vlsi architecture: Past, present, and future. `http://cva.stanford.edu/people/dally/ARVLSI99.ppt`, March 1999. Computer Systems Laboratory, Stanford University.

Conitec Datasystems. 3d game studio. `http://www.3dgamestudio.com`, June 2007.

Doug DeCarlo, Adam Finkelstein, and Szymon Rusinkiewicz. Interactive rendering of suggestive contours with temporal coherence. In *NPAR '04: Proceedings of the 3rd International Symposium on Non-photorealistic Animation and Rendering*, pages 15–145, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-887-3.

Doug DeCarlo, Adam Finkelstein, Szymon Rusinkiewicz, and Anthony Santella. Suggestive contours for conveying shape. *ACM Transactions on Graphics*, 22(3):848–855, 2003. ISSN 0730-0301.

Kelly Dempski. *Real Time Rendering Tricks and Techniques in DirectX*. Thomson Course Technology PTR, Boston, MA, USA, 1st edition, March 2002. ISBN 1931841276.

George Eckel. *Cosmo3D Programmer's Guide*. Silicon Graphics, Mountain View, CA, USA, 1998.

George Eckel and Ken Jones. *OpenGL Performer Programmer's Guide Version 3.2*. Silicon Graphics, Mountain View, CA, USA, 2004.

Interactive Effects. Amazon paint. `http://www.ifx.com/amazon`, March 2007.

Gershon Elber. Interactive line art rendering of freeform surfaces. *Computer Graphics Forum*, 18(3): 1–12, 1999.

Wolfgang Engel. *Programming Vertex & Pixel Shaders*. Charles River Media, 1st edition, September 2004. ISBN 1584503491.

Randima Fernando. Trends in gpu evolution. In *Eurographics 2004 Industrial Seminars 2, Next Generations Graphics Hardware*, Grenoble, France, August 2004. Presentation slides.

David Flanagan. *Motif Tools: Application Design and Development with XT, Motif and the Motif Tools Library*. O'Reilly, September 1994. ISBN 1-565-92044-9.

James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley Publishing Co., Reading, MA, USA, 2nd edition, 1990.

Ron Fosner. *Real-Time Shader Programming*. Morgan Kaufmann, December 2002. ISBN 1558608532.

GarageGames. Torque game engine. `http://www.garagegames.com`, June 2007.

Tom Gaskings. *PHIGS Programming Manual*. O'Reilly & Associates, 1992.

Nikolaus Gebhardt. Irrlicht engine. `http://irrlicht.sourceforge.net`, June 2007.

Michael Gleicher. Image snapping. In *Proceedings of SIGGRAPH 1995*, pages 183–190, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-701-4.

Jack Goldfeather and Victoria Interrante. A novel cubic-order algorithm for approximating principal direction vectors. *ACM Transactions on Graphics*, 23(1):45–63, 2004. ISSN 0730-0301.

Henri Gouraud. Continuous shading of curved surfaces. *IEEE Transactions on Computers*, 20(6): 623–629, 1971.

Naga K. Govindaraju, Nikunj Raghuvanshi, Michael Henson, Ming Lin, and Dinesh Manocha. Gpusort: High performance sorting using graphics processors. `http://gamma.cs.unc.edu/GPUSORT`, 2003. Department of Computer Science, UNC Chapel Hill.

GPGPU. General-purpose computation using graphics hardware. `http://www.gpgpu.org/`, June 2007.

Silicon Graphics. Irix operating system. `http://www.sgi.com/products/software/irix`, April 2007.

Jens Gravesen and Michael Ungstrup. Constructing invariant fairness measures for surfaces. *Advances in Computational Mathematics*, 17:67–88, 2002.

Simon Green. High dynamic range and other fun shader tricks. In *Game Developers Conference*, March 2002. Presentation slides.

B. Hamann. Curvature approximation for triangulated surfaces. *Computing Suppl.*, 8:139–153, 1993.

Chris Hand. A survey of 3d interaction techniques. *Computer Graphics Forum*, 16(5):269–281, 1997.

Patrick Hanrahan and Paul Haeberli. Direct wysiwyg painting and texturing on 3d shapes. In *SIGGRAPH '90: Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, pages 215–223, Dallas, TX, USA, 1990. ACM Press. ISBN 0-201-50933-4.

Patrick Hanrahan and Jim Lawson. A language for shading and lighting calculations. *ACM SIGGRAPH Computer Graphics*, 24(4), 1990.

Chris Hecker. Behind the screen: An open letter to microsoft. In *Game Developer Magazine*, pages 16–21. CMP Technology, April 1997.

Right Hemisphere. Deep creator and deep paint 3d. `http://www.righthemisphere.com`, March 2007.

HP. Hewlett-packard development company. `http://www.hp.com`, April 2007.

Scott E. Hudson. Adaptive semantic snapping - a technique for feedback at the lexical level. In *Proceedings of CHI 1990*, pages 65–70. ACM, ACM Press / ACM SIGGRAPH, April 1990.

IBM. `http://www.ibm.com`, April 2007.

Chris Insinger. Silicon valley siggraph notes: Apis of the fahrenheit initiative. `http://silicon-valley.siggraph.org/MeetingNotes/Fahrenheit.html`, September 1998. Notes by Jenny Dana.

Intel. Intel corporation. `http://www.intel.com`, April 2007.

John Isidoro, Alex Vlachos, and Chris Brennan. Rendering ocean water. In Wolfgang Engel, editor, *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*. Wordware Publishing, 2002. ISBN 1556220413.

ISO. Iso/iec 7942:1985 information technology – computer graphics – graphical kernel system (gks) – part 1: Functional description, part 2: Ndc metafile, part 3: Audit, and part 4: Archive, 1985.

Khronos. The khronos group. `http://www.khronos.org`, April 2007.

Emmett Kilgariff and Randima Fernando. The geforce 6 series gpu architecture. In Matt Pharr and Randima Fernando, editors, *GPU Gems 2*, pages 471–491. Addison-Wesley, 2005.

Felix Kälberer, Konrad Polthier, Ulrich Reitebuch, and Max Wardetzky. Freelence: Compressing triangle meshes using geometric information. Technical report, Zuse Institute Berlin, July 2004.

G. Kortun. *Reflectance Spectroscopy*. Springer-Verlag, New York, NY, USA, 1969.

Venkat Krishnamurthy and Marc Levoy. Fitting smooth surfaces to dense polygon meshes. In *Proceedings of SIGGRAPH 1996*, pages 313–324, New York, NY, USA, 1996. ACM Press. ISBN 0-89791-746-4.

Yuri Kryachko. Using vertex texture displacement for realistic water rendering. In Matt Pharr and Randima Fernando, editors, *GPU Gems 2*, chapter 18, pages 283–294. Addison-Wesley, March 2005. ISBN 0321335597.

Carsten Lange and Konrad Polthier. Anisotropic smoothing of point sets. *Comput. Aided Geom. Des.*, 22(7):680–692, 2005. ISSN 0167-8396.

Aaron Lefohn. Data formatting and addressing. In *SIGGRAPH 2004 GPGPU Course: Effective GPGPU Programming, Tricks and Traps*, August 2004. Presentation slides.

Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics*. Charles River Media, 2nd edition, November 2003. ISBN 1-584-50277-0.

Eric Lengyel. Bitangent vector – from wolfram mathworld. `http://mathworld.wolfram.com/BitangentVector.html`, June 2007.

Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Proceedings of SIGGRAPH 2001*, pages 149–158, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-374-X.

Syd Logan. *Gtk+ Programming in C*. Prentice Hall Ptr, 1st edition, August 2001. ISBN 0-130-14264-6.

David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. Gpgpu: General purpose computation on graphics hardware. In *SIGGRAPH '04: Proceedings of the conference on SIGGRAPH 2004 course notes*, New York, NY, USA, 2004a. ACM Press.

David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. Gpgpu: general purpose computation on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, page 33, New York, NY, USA, 2004b. ACM Press.

Matrox. Matrox graphics. `http://www.matrox.com/mga`, April 2007.

Nelson Max. Weights for computing vertex normals from facet normals. *Journal of Graphics Tools*, 4(2):1–6, 1999. ISSN 1086-7651.

Patricia McLendon. *Graphics Library Programming Guide*. Silicon Graphics, Mountain View, CA, USA, 1992.

Mark Meyer, Mathieu Desbrun, Peter Schröder, and Alan H. Barr. Discrete differential-geometry operators for triangulated 2-manifolds. In Hans-Christian Hege and Konrad Polthier, editors, *Proceedings of Visualization and Mathematics III*, pages 35–57, Heidelberg, Germany, 2003. Springer-Verlag.

Microsoft. *DirectX June 2006 SDK Programmer's Reference*. Microsoft Corporation, Redmond, WA, USA, June 2006.

Microsoft. `http://msdn.microsoft.com/directx`, April 2007.

Pick sample, DirectX 9.0 SDK Sample Browser. Microsoft Corporation, December 2005.

Brad A. Myers. User interface software tools. *ACM Transactions on Computer-Human Interaction*, 2(1):64–103, 1995. ISSN 1073-0516.

Gregory M. Nielson and Jr. Dan R. Olsen. Direct manipulation techniques for 3d objects using 2d locator devices. In *SI3D '86: Proceedings of the 1986 Workshop on Interactive 3D Graphics*, pages 175–182, New York, NY, USA, 1987. ACM Press. ISBN 0-89791-228-4.

NVIDIA. Comparison of nvidia graphics processing units. `http://en.wikipedia.org/wiki/Comparison_of_NVIDIA_Graphics_Processing_Units`, April 2007a.

NVIDIA. Nvidia corporation. http://www.nvidia.com, April 2007b.

Yutaka Ohtake, Alexander Belyaev, and Hans-Peter Seidel. Ridge-valley lines on meshes via implicit surface fitting. *ACM Transactions of Graphics*, 23(3):609–612, 2004. ISSN 0730-0301.

Dan R. Olsen. *Developing User Interfaces*. Morgan Kaufmann, San Francisco, CA, USA, 1998. ISBN 1-55860-418-9.

OpenSceneGraph. http://www.openscenegraph.org, April 2007.

OpenSG. http://opensg.vrsource.org, April 2007.

Michael Oren and Shree Nayar. Generalization of the lambertian model and implications for machine vision. *International Journal of Computer Vision*, 14(3), 1992.

Ken Perlin. An image synthesizer. In *SIGGRAPH '85: Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, pages 287–296, New York, NY, USA, 1985. ACM Press. ISBN 0-89791-166-0.

Jeffrey S. Pierce and Randy Pausch. Specifying interaction surfaces using interaction maps. In *Proceedings of the 2003 Symposium on Interactive 3D Graphics*, pages 189–192, Monterey, CA, USA, 2003. ACM Press. ISBN 1-58113-645-5.

Pixar. http://www.pixar.com/, April 2007.

Pixologic. Zbrush. http://pixologic.com, March 2007.

Fábio Policarpo, Manuel M. Oliveira, and João L. D. Comba. Real-time relief mapping on arbitrary polygonal surfaces. In *SI3D '05: Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, pages 155–162, Washington, DC, USA, April 2005. ACM Press. ISBN 1-59593-013-2.

Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. Real-time hatching. In *SIGGRAPH '01: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, page 581, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-374-X.

Timothy J. Purcell. *Ray Tracing on a Stream Processor*. PhD thesis, Stanford University, March 2004. http://graphics.stanford.edu/papers/tpurcell_thesis.

Timothy J. Purcell, Ian Buck, William R. Mark, and Patrick Hanrahan. Ray tracing on programmable graphics hardware. In *Proceedings of SIGGRAPH 2002*, pages 703–712, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-521-1.

L. G. Roberts. *Machine perception of 3-D solids*. PhD thesis, Massachusetts Institute of Technology, 1963.

Randi Rost. *OpenGL++ ARB Meeting Notes*. OpenGL Architecture Review Board, February 1997.

Szymon Rusinkiewicz. Estimating curvatures and their derivatives on triangle meshes. In *3DPVT '04: Proceedings of the 3D Data Processing, Visualization, and Transmission, 2nd International Symposium on (3DPVT'04)*, pages 486–493, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2223-8.

Szymon Rusinkiewicz. Trimesh2 triangle mesh library. `http://www.cs.princeton.edu/gfx/proj/trimesh2`, 2006.

Szymon Rusinkiewicz, Doug DeCarlo, Adam Finkelstein, and Anothony Santella. Suggestive contour gallery. `http://www.cs.princeton.edu/gfx/proj/sugcon/models/`, June 2007.

S3. S3 graphics co., ltd. `http://www.s3graphics.com`, April 2007.

Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. In *Proceedings of SIGGRAPH 1990*, pages 197–206, Dallas, TX, USA, 1990a. ACM Press. ISBN 0-201-50933-4.

Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 197–206, New York, NY, USA, 1990b. ACM Press. ISBN 0-201-50933-4.

Yoichi Sato, Mark D. Wheeler, and Katsushi Ikeuchi. Object shape and reflectance modeling from observation. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 379–387, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co. ISBN 0-89791-896-7.

SGI. Sgi - products: Software: Apis. `http://www.sgi.com/products/software/apis.html`, June 2007a.

SGI. Silicon graphics. `http://www.sgi.com`, April 2007b.

Ben Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69, 1983.

Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2*. Addison-Wesley, 2005. ISBN 0-321-33573-2.

OpenGL Performer: Real-Time 3D Rendering for High-Performance and Interactive Graphics Applications. Silicon Graphics, Mountain View, CA, USA, 2005.

Paul S. Strauss. Iris inventor, a 3d graphics toolkit. In *OOPSLA '93: Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 192–200, New York, NY, USA, 1993. ACM Press. ISBN 0-89791-587-9.

Sun. Sun microsystems. `http://www.sun.com/`, April 2007a.

Sun. Sun microsystems: Java 3d parent project. `https://java3d.dev.java.net`, April 2007b.

Ivan E. Sutherland. Sketchpad: A man-machine graphics communication system. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 329–346, 1963.

Xiaojing Tang and Gady Agam. A sampling framework for accurate curvature estimation in discrete surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):573–583, 2005. ISSN 1077-2626.

Natalya Tatarchuk. Dynamic parallax occlusion mapping with approximate soft shadows. In *SI3D '06: Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, pages 63–69, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-295-X.

G. Taubin. Estimating the tensor of curvature of a surface from a polyhedral approximation. In *ICCV '95: Proceedings of the Fifth International Conference on Computer Vision*, page 902, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-7042-8.

Grit Thürmer and Charles A. Wüthrich. Computing vertex normals from polygonal facets. *Journal of Graphics Tools*, 3(1):43–46, 1998. ISSN 1086-7651.

SkinnedMesh Pick Problem. Usenet Newsgroup Thread, February 2002. `microsoft.public.win32.programmer.directx.graphics`.

Determining Which Skinning Method to Use. Usenet Newsgroup Thread, May 2004. `microsoft.public.win32.programmer.directx.graphics`.

Andries van Dam. Phigs+ functional description, revision 3.0. *Computer Graphics*, 22(3):125–218, July 1988.

Ir. Mfa Koert van Mensvoort. What you see is what you feel: Exploiting the dominance of the visual over the haptic domain to simulate force-feedback with cursor displacements. In *DIS '02: Proceedings of the Conference on Designing Interactive Systems*, pages 345–348, London, England, 2002. ACM Press. ISBN 1-58113-515-7.

Lifeng Wang, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. View-dependent displacement mapping. *ACM Transactions on Graphics*, 22(3):334–339, 2003. ISSN 0730-0301.

Hank Weghorst, Gary Hooper, and Donald P. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, 3(1):52–69, 1984. ISSN 0730-0301.

Ulrich Weidenbacher, Pierre Bayerl, Roland Fleming, and Heiko Neumann. Extracting and depicting the 3d shape of specular surfaces. In *APGV '05: Proceedings of the 2nd Symposium on Applied Perception in Graphics and Visualization*, pages 83–86, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-139-2.

Terry Welsh. Parallax mapping. In Wolfgang Engel, editor, *ShaderX3: Advanced Rendering with DirectX and OpenGL*. Charles River Media, November 2004.

W. WM. Wendlandt and H. G. Hecht. Reflectance spectroscopy. *Chemical Analysis*, 21, 1966.

Josie Wernecke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor, Release 2*. Addison-Wesley, 1st edition, 1994. ISBN 0-201-62495-8.

Geir Westgaard and Horst Nowacki. Construction of fair surfaces over irregular meshes. In *SMA '01: Proceedings of the Sixth ACM Symposium on Solid Modeling and Applications*, pages 88–98, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-366-9.

Sven Woop, Jörg Schmittler, and Philipp Slusallek. Rpu: A programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics*, 24(3):434–444, 2005. ISSN 0730-0301.

S. T. Wu, M. Abrantes, D. Tost, and H. C. Batagelo. Picking and snapping for 3d input devices. In *XVI Brazilian Symposium on Computer Graphics and Image Processing*, pages 140–147, October 2003.

Kwan-Hee Yoo and Jong-Sung Ha. Geometric snapping for 3d meshes. In *International Conference on Computational Science*, pages 90–97, Krakow, Poland, June 2004.

Robert C. Zeleznik, D. Brookshire Conner, Matthias M. Wloka, Daniel G. Aliaga, Nathan T. Huang, Philip M. Hubbard, Brian Knep, Henry Kaufman, John F. Hughes, and Andries van Dam. An object-oriented framework for the integration of interactive animation techniques. *Computer Graphics*, 25(4):105–112, 1991.

Cyril Zeller. Cloth simulation on the gpu. In *Proceedings of SIGGRAPH 2005*, March 2005. Presentation slides.

Apêndice A

Shaders de estimativa de elementos de geometria diferencial de primeira ordem

A seguir apresentamos três *shaders* de fragmentos em linguagem HLSL (*High Level Shading Language*) [Microsoft, 2006], que compõem o procedimento utilizado para o cálculo de bases tangentes alinhadas às coordenadas de textura, conforme exposto na seção 4.2. Os *shaders* requerem GPUs compatíveis com o modelo de *shader* 3.0 ou posterior.

O cálculo de bases tangentes para cada vértice é realizado em dois passos de renderização que utilizam a GPU como um processador de propósito geral. Entretanto, por completude, mostramos aqui três passos de renderização, sendo que o primeiro passo corresponde, na arquitetura proposta, à execução do estágio de modificação de atributos de vértices. Em especial, a função `UpdateVPos()` é utilizada no primeiro passo para atualizar o atributo de posição de cada vértice e gravar o resultado em um novo mapa de atributo de posições correspondente ao objeto de amostragem `sUpdVPos`. O segundo passo utiliza a função `ComputeFaceNTB()` para calcular as bases tangentes de cada face. Os resultados são gravados nas texturas indicadas pelo objeto de amostragem `sFaceN` (vetor normal por face), `sFaceT` (vetor tangente por face) e `sFaceB` (vetor bitangente por face). O último passo de renderização utiliza a função `ComputeVertexTB()` para calcular a média das as bases tangentes de cada vértice e ortonormalizá-las em seguida. O resultado é gravado em um mapa de normais no formato RGB e em um mapa de tangentes no formato $RGB\alpha$, onde α indica qual regra de mão deve ser utilizada para obter o vetor bitangente.

```
////////////////////////////////////////////////////////////////
// Converte um índice 1D (iIdx) para um índice 2D de uma
// textura (coordenadas entre 0 e 1).
// vCons = {1/w, 1/(w*h)} onde w e h são a largura e
// altura do mapa de textura.
```

```
////////////////////////////////////////////////////////////////////
float2 Idx1DTo2D( float iIdx, float2 vCons )
{
  return float2( iIdx, iIdx ) * vCons;
}

////////////////////////////////////////////////////////////////////
// Constantes definidas pela aplicação e utilizadas como segundo
// parâmetro de Idx1DTo2D( ).
////////////////////////////////////////////////////////////////////
float2 g_vVtCons;   // Utilizado em mapas de atributos de vértices
float2 g_vFaceCons; // Utilizado em mapas de atributos de faces
float2 g_vAdjCons;  // Utilizado com o mapa de relações de adjacência

////////////////////////////////////////////////////////////////////
// Objetos de amostragem de textura.
////////////////////////////////////////////////////////////////////

sampler sVPos;    // Mapa de posições dos vértices
sampler sUpdVPos; // Mapa de posições atualizadas dos vértices
sampler sVTex;    // Mapa de coordenadas de textura dos vértices
sampler sFace;    // Mapa de faces
sampler sAdjIdx;  // Mapa de índices de adjacência
sampler sAdj;     // Mapa de relações de adjacência
sampler sFaceN;   // Mapa de normais às faces
sampler sFaceT;   // Mapa de tangentes às faces
sampler sFaceB;   // Mapa de bitangente às faces

////////////////////////////////////////////////////////////////////
// 1o. passo: Atualiza as posições dos vértices de acordo com a
// função de deformação.
////////////////////////////////////////////////////////////////////
float4 UpdateVPos( float2 vTex ) : COLOR
{
  // Lê a posição do vértice
  float4 vPos = tex2D( sVPos, vTex );
  // Chama a função de deformação definida pela aplicação
  // Obs.: A função não é apresentada nessa listagem
```

```
  vPos = Deform( vPos );
  // Grava o resultado no mapa de posições atualizadas dos vértices
  return vPos;
}

////////////////////////////////////////////////////////////////////
// 2o. passo: Calcula as bases tangentes para cada face.
////////////////////////////////////////////////////////////////////
struct PS_FACENTBOUT { float3 vNTB[3] : COLOR; };
PS_FACENTBOUT ComputeFaceNTB( float2 vTex )
{
  PS_FACENTBOUT Out;

  // Lê os índices dos três vértices que compõem a face
  float3 vVtIdx = tex2D( sFace, vTex );

  // Lê as posições atualizadas de vértices
  float3 vVtUV1 = Idx1DTo2D( vVtIdx.x, g_vVtCons );
  float3 vVtUV2 = Idx1DTo2D( vVtIdx.y, g_vVtCons );
  float3 vVtUV3 = Idx1DTo2D( vVtIdx.z, g_vVtCons );
  float3 vV1 = tex2D( sUpdVPos, vVtUV1 );
  float3 vV2 = tex2D( sUpdVPos, vVtUV2 );
  float3 vV3 = tex2D( sUpdVPos, vVtUV3 );

  // Calcula normal à face e grava do mapa de normais às faces
  float3 vDV1 = vV2 - vV1;
  float3 vDV2 = vV3 - vV1;
  Out.vNTB[0] = cross( vDV1, vDV2 );

  // Lê as coordenadas de textura de cada vértice
  float2 vW1 = tex2D( sVTex, vVtUV1 );
  float2 vW2 = tex2D( sVTex, vVtUV2 );
  float2 vW3 = tex2D( sVTex, vVtUV3 );

  // Calcula tangente e bitangente à face e grava os resultados
  // em duas texturas
  float2 vDW1 = vW2 - vW1;
  float2 vDW2 = vW3 - vW1;
```

```
  float fR = 1.0f / ( vDW1.x*vDW2.y - vDW2.x*vDW1.y );

  float3 vT = float3( vDW2.y*vDV1.x - vDW1.y*vDV2.x,
                      vDW2.y*vDV1.y - vDW1.y*vDV2.y,
                      vDW2.y*vDV1.z - vDW1.y*vDV2.z );

  float3 vB = float3( vDW2.x*vDV1.x - vDW1.x*vDV2.x,
                      vDW2.x*vDV1.y - vDW1.x*vDV2.y,
                      vDW2.x*vDV1.z - vDW1.x*vDV2.z );

  Out.vNTB[1] = vT * fR;
  Out.vNTB[2] = vB * fR;

  return Out;
}

////////////////////////////////////////////////////////////////////
// 3o. passo: Calcula as bases tangentes para cada vértice.
// Cada vértice vai conter a média das bases tangentes das faces
// incidentes. As bases tangentes são ortonormalizadas no final.
////////////////////////////////////////////////////////////////////
struct PS_VERTEXNTOUT { float3 vNT[2] : COLOR; };
PS_VERTEXNTOUT ComputeVertexTB( float2 vTex )
{
  PS_VERTEXNTOUT Out;

  float2 fIdx2;
  float3 vSumN, vSumT, vSumB;
  vSumN = vSumT = vSumB = float3( 0.0, 0.0, 0.0 );

  // Lê mapa de índices de adjacência
  float3 vAdjIdx = tex2D( sAdjIdx, vTex );

  // Soma as normais e tangentes das faces adjacentes
  for( int iCt = 0; iCt < vAdjIdx.y; iCt++ )
  {
    // Obtém o índice da face atual no mapa de adjacência
```

```
    fIdx2 = Idx1DTo2D( vAdjIdx.x + iCt, g_vAdjCons );
    float fFIdx = tex2Dlod( sAdj, float4(fIdx2,0,0) );

    // Lê a normal à face
    fIdx2 = Idx1DTo2D( fFIdx, g_vFaceCons );
    float3 vFN = tex2Dlod( sFaceN, float4(fIdx2,0,0) );

    // Lê tangente e bitangente à face
    float3 vFT = tex2Dlod( sFaceT, float4(fIdx2,0,0) );
    float3 vFB = tex2Dlod( sFaceB, float4(fIdx2,0,0) );

    // Acumula a contribuição desta face
    vSumN += vFN;
    vSumT += vFT;
    vSumB += vFB;
  }

  // Grava o vetor normal ao vértice
  float3 vVtN = normalize( vSumN );
  Out.vNT[0] = float4( vVtN, 1.0f );

  // Ortonormaliza a base tangente usando Gram-Schmidt
  float3 vRes = normalize( vSumT - vVtN * dot( vVtN, vSumT ) );
  float4 vVtT = float4( vRes, 1.0 );

  // Calcula regra de mão utilizada
  if( dot( cross( vVtN, vSumT ), vSumB ) < 0.0f )
    vVtT.w = 0.0;

  // Grava no mapa de tangentes aos vértices
  Out.vNT[1] = vVtT;

  return Out;
}
```

Apêndice B

Shaders de estimativa de elementos de geometria diferencial de segunda e terceira ordem

Neste apêndice apresentamos dois *shaders* de fragmentos utilizados para estimar elementos de geometria diferencial de segunda e terceira ordem segundo as abordagens descritas na seção 4.2. Os *shaders* requerem GPUs compatíveis com o modelo de *shader* 3.0 [Microsoft, 2006] ou posterior.

O cálculo de elementos de geometria diferencial de segunda ordem é realizado pelo *shader* de fragmentos `Tensor2()` e produz como resultado os três componentes do tensor de curvatura, as direções principais e curvaturas principais. Essa função assume que as normais aos vértices já foram calculadas e que a geometria já foi atualizada pelas deformações de vértices (veja o apêndice A).

O *shader* de fragmentos `Tensor3()` calcula os quatro componentes do tensor de derivada de curvatura. Sua execução assume que a função `Tensor2()` foi executada previamente para calcular os mapas de texturas que contém os resultados das direções e curvaturas principais. Essas texturas são então acessadas pelos objetos de amostragem `sP1K1` e `sP2K2`.

Para simplificar a apresentação da listagem a seguir, a função `ldltsv()`, utilizada para resolver um sistema de equações lineares através da decomposição LDL', aceita um parâmetro N que indica a dimensão da matriz. Para evitar o uso de instruções de fluxo de controle dinâmico e assim aumentar a eficiência, esta função pode ser dividida em duas: Uma com o parâmetro fixado em três, para seu uso com `Tensor2()`, e outra com o parâmetro fixado em quatro, para seu com `Tensor3()`.

```
////////////////////////////////////////////////////////////////
// Converte um índice 1D (iIdx) para um índice 2D de uma
// textura (coordenadas entre 0 e 1).
// vCons = {1/w, 1/(w*h)} onde w e h são a largura e
```

```
// altura do mapa de textura.
//////////////////////////////////////////////////////////////////////
float2 Idx1DTo2D( float iIdx, float2 vCons ) {
  return float2( iIdx, iIdx ) * vCons;
}

//////////////////////////////////////////////////////////////////////
// Constantes definidas pela aplicação e utilizadas como segundo
// parâmetro de Idx1DTo2D( ).
//////////////////////////////////////////////////////////////////////
float2 g_vAdjCons; // Utilizado em mapas de adjacências de vértices
float2 g_vVtCons;  // Utilizado em mapas de atributos de vértices

//////////////////////////////////////////////////////////////////////
// Semânticas de entrada do shader de vértices.
//////////////////////////////////////////////////////////////////////
struct VS_IN {
  float4 vPos : POSITION;  // Posição transformada (XYZ+RHW).
  float2 vTex : TEXCOORD0; // Coordenadas de textura (UV).
};

//////////////////////////////////////////////////////////////////////
// Semânticas de saída do shader de fragmentos para Tensor2().
//////////////////////////////////////////////////////////////////////
struct PS_OUT1 {
  // vColor[0]: 2a. forma fundamental. RGB=efg
  // vColor[1]: 1a. dir. principal (RGB) e curv. principal k1 (A)
  // vColor[2]: 2a. dir. principal (RGB) e curv. principal k2 (A)
  float4 vColor[3] : COLOR;
};

//////////////////////////////////////////////////////////////////////
// Semânticas de saída do shader de fragmentos para Tensor3().
//////////////////////////////////////////////////////////////////////
struct PS_OUT2 {
  // Tensor de derivada de curvatura (4 valores)
  float4 vColor : COLOR;
};
```

```
sampler sVPos;   // Mapa de posições dos vértices
sampler sVN;     // Mapa de normais aos vértices
sampler sAdjIdx; // Mapa de índices de adjacência
sampler sAdj;    // Mapa de relações de adjacência
sampler sP1K1;   // Mapa com direção e curvatura principal 1
sampler sP2K2;   // Mapa com direção e curvatura principal 2

////////////////////////////////////////////////////////////////////
// Realiza a decomposição LDL' de uma matriz NxN simétrica positiva
// definida e resolve Ax=B.
////////////////////////////////////////////////////////////////////
float3 ldltsv( float3x3 w, float3 m, float N ) {
  // Decompõe matriz e sobrescreve matriz triangular inferior.
  float3 d;
  float2 v;
  int i;
  for( i = 0; i < N; i++ )
  {
    for( int k = 0; k < i; k++ ) v[k] = w[i][k] * d[k];
    for( int j = i; j < N; j++ )
    {
      float sum = w[i][j];
      for( int k = 0; k < i; k++ )
        sum -= v[k] * w[j][k];
      if( i == j )
        d[i] = 1.0f / sum;
      else
        w[j][i] = sum;
    }
  }

  // Resolve Ax=B após a decomposição LDL'
  for( i = 0; i < N; i++ )
  {
    float sum = m[i];
    for( int k = 0; k < i; k++ )
      sum -= w[i][k] * m[k];
```

```
    m[i] = sum * d[i];
  }
  for( i = N-1; i >= 0; i-- )
  {
    float sum = 0;
    for( int k = i + 1; k < N; k++ )
      sum += w[k][i] * m[k];
    m[i] -= sum * d[i];
  }

  return m;
}

////////////////////////////////////////////////////////////////////
// Calcula elementos de segunda ordem.
////////////////////////////////////////////////////////////////////
PS_OUT1 Tensor2( VS_IN In ) {
  PS_OUT1 Out;

  // Amostra índices de adjacência
  // .z) Índice do 1o. vértice adjacente no mapa de vértices adjacentes
  // .w) Número de vértices adjacentes
  float4 fAdj = tex2D( sAdjIdx, In.vTex );
  if( fAdj.z == -1.0 ) discard;

  // Índices UV do primeiro vértice da vizinhaça
  float2 vAdjUVNei  = I1DTo2D( fAdj.z, g_vAdjCons );
  float  fIdxNei    = tex2D( sAdj, vAdjUVNei ).x;
  float2 vUVNei     = I1DTo2D( fIdxNei, g_vVtCons );

  // Calcula uma base tangente ortogonal inicial
  float3 vPos       = tex2D( sVPos, In.vTex );
  float3 vNormal    = tex2D( sVN,   In.vTex );
  float3 vPosNei    = tex2D( sVPos, vUVNei  );
  float3 vNormalNei = tex2D( sVN,   vUVNei  );
  float3 vPDir1     = normalize( cross( (vPosNei - vPos), vNormal ) );
  float3 vPDir2     = cross( vNormal, vPDir1 );
```

```
float3 m = 0;
float3x3 w = 0;

// Para cada vértice da vizinhança 1-ring:
int iCt = 0;
while( iCt < fAdj.w )
{
  // Obtém índice do vértice do mapa de adjacência
  vAdjUVNei  = I1DTo2D( fAdj.z + iCt, g_vAdjCons );
  fIdxNei    = tex2Dlod( sAdj, float4( vAdjUVNei,0,0 ) ).x;

  // Obtém atributos do vértice
  vUVNei     = I1DTo2D( fIdxNei, g_vVtCons );
  vPosNei    = tex2Dlod( sVPos, float4( vUVNei,0,0 ) );
  vNormalNei = tex2Dlod( sVN,   float4( vUVNei,0,0 ) );

  float3 e = vPosNei - vPos;
  float  u = dot( e, vPDir1 );
  float  v = dot( e, vPDir2 );
  w[0][0] += u*u;
  w[0][1] += u*v;
  w[2][2] += v*v;
  float3 dn = vNormalNei - vNormal;
  float dnu = dot( dn, vPDir1 );
  float dnv = dot( dn, vPDir2 );
  m[0] += dnu*u;
  m[1] += dnu*v + dnv*u;
  m[2] += dnv*v;

  iCt++;
}

w[1][1] = w[0][0] + w[2][2];
w[1][2] = w[0][1];

m = ldltsv( w, m, 3 );

// Grava tensor de curvatura
```

```
  Out.vColor[0] = float4( m, 0 );

  // Rotação de Jacobi para diagonalizar
  float ku = m[0], kuv = m[1], kv = m[2];
  float c = 1, s = 0, tt = 0;
  if( kuv != 0.0f )
  {
    float h = 0.5f * (kv - ku) / kuv;
    tt = (h < 0.0f) ? 1.0f / (h - (float) sqrt(1.0f + h*h)) :
         1.0f / (h + (float) sqrt(1.0f + h*h));
    c = 1.0f / (float) sqrt(1.0f + tt*tt);
    s = tt * c;
  }

  float fK1 = ku - tt * kuv,
        fK2 = kv + tt * kuv;

  float3 pd1, pd2;
  if( abs( fK1 ) >= abs( fK2 ) )
    pd1 = c * vPDir1 - s * vPDir2;
  else
  {
    float fTemp = fK1; fK1 = fK2; fK1 = fTemp;
    pd1 = s * vPDir1 + c * vPDir2;
  }
  pd2 = cross( vNormal, pd1 );

  // Grava direção e curvatura principal mínima
  Out.vColor[1] = float4( pd1, fK1 );

  // Grava direção e curvatura principal máxima
  Out.vColor[2] = float4( pd2, fK2 );

  return Out;
}

//////////////////////////////////////////////////////////////////////
// Calcula elementos de terceira ordem.
```

```
////////////////////////////////////////////////////////////////////
PS_OUT2 Tensor3( VS_IN In ) {
  PS_OUT2 Out;

  // Amostra índices de adjacência
  // .z) Índice do 1o. vértice adjacente no mapa de vértices adjacentes
  // .w) Número de vértices adjacentes
  float4 fAdj = tex2D( sAdjIdx, In.vTex );
  if( fAdj.z == -1.0 ) discard;

  float3 vPos   = tex2D( sVPos, In.vTex );
  float4 vpk1   = tex2D( sP1K1, In.vTex );
  float4 vpk2   = tex2D( sP2K2, In.vTex );

  float3 vPDir1 = vpk1.xyz,
         vPDir2 = vpk2.xyz;
  float2 vCurv  = float2( vpk1.w, vpk2.w );
  float2 vCurvNei;

  float4 m = 0;
  float4x4 w = 0;

  int iCt = 0;
  while( iCt < fAdj.w )
  {
    // Obtém índice do vértice do mapa de adjacência
    float2 vAdjUVNei = I1DTo2D( fAdj.z + iCt, g_vAdjCons );
    float  fIdxNei   = tex2Dlod( sAdj, float4( vAdjUVNei,0,0 ) ).x;

    // Obtém atributos do vértice
    float2 vUVNei    = I1DTo2D( fIdxNei, g_vVtCons );
    float3 vPosNei   = tex2Dlod( sVPos, float4( vUVNei,0,0 ) );
    vCurvNei.x       = tex2Dlod( sP1K1, float4( vUVNei,0,0 ) ).w;
    vCurvNei.y       = tex2Dlod( sP2K2, float4( vUVNei,0,0 ) ).w;

    float3 fcurv_i = float3( vCurv.x,    0, vCurv.y    ),
           fcurv_n = float3( vCurvNei.x, 0, vCurvNei.y );
```

```
    // Variação da curvatura ao longo da aresta até o vértice
    float3 dfcurv = fcurv_n - fcurv_i;
    float3 e = vPosNei - vPos;

    float u = dot( e, vPDir1 );
    float v = dot( e, vPDir2 );
    float u2 = u*u, v2 = v*v, uv = u*v;
    w[0][0] += u2;
    w[0][1] += uv;
    w[3][3] += v2;
    m[0] += u*dfcurv[0];
    m[1] += v*dfcurv[0];
    m[2] += u*dfcurv[2];
    m[3] += v*dfcurv[2];

    iCt++;
  }

  w[1][1] = 2.0f * w[0][0] + w[3][3];
  w[1][2] = 2.0f * w[0][1];
  w[2][2] = w[0][0] + 2.0f * w[3][3];
  w[2][3] = w[0][1];

  m = ldltsv( w, m, 4 );

  // Grava elementos do tensor de derivada de curvatura
  Out.vColor = m;

  return Out;
}
```

Apêndice C

Interface de programação

Com base na arquitetura proposta no capítulo 5, e utilizando as técnicas de estimativa de propriedades de geometria diferencial na GPU apresentadas no capítulo 4, implementamos uma biblioteca de funções de suporte a tarefas de manipulação direta 3D em geometria modificada em *hardware* gráfico programável. A biblioteca encapsula todo o fluxo de processamento da arquitetura de interação de forma transparente à aplicação, e pode ser utilizada para a prototipação rápida de ferramentas de manipulação direta em aplicações de interação 3D que utilizem geometria modificada em tempo real no processador de vértices da GPU.

A biblioteca é implementada em C++ e pode utilizar as APIs gráficas OpenGL e Direct3D de acordo com a escolha da aplicação. As funções de suporte a tarefas de interação estão organizadas em duas classes C++:

- Classe `CIntManager`: responsável pelo gerenciamento dos estados que afetam todos os modelos sobre os quais se deseja que os atributos sejam gerados para cada *pixel*. Em particular, cada instância da classe gerencia um conjunto de modelos que geram o mesmo conjunto de tipos de atributos.

  As funções de interação incluídas em `CIntManager` incluem as funções de configuração dos atributos geométricos ou não geométricos que serão calculados pela arquitetura, bem como as funções de mapeamento de semântica desses atributos. A classe também inclui funções de configuração da região de interesse do cálculo dos atributos, e a função que dispara o processamento dos atributos na GPU e codifica-os em *buffers* de renderização não visíveis. A função de decodificação dos atributos calculados também está incluída nesta classe.

- Classe `CIntObj`: responsável pelo gerenciamento dos estados relacionados a um modelo específico utilizado na arquitetura. Cada instância desta classe representa o estado de apenas um modelo da cena sobre o qual se deseja que os atributos sejam calculados.

Cada instância de `CIntManager` gerencia um conjunto de instâncias de `CIntObj`. Tais instâncias de `CIntObj` correspondem a modelos geométricos que vão gerar um mesmo conjunto de tipos de atributos. Por outro lado, cada instância pode estar associada a um tipo diferente de *shader* de modificação de atributos de vértices e *shader* de modificação de fragmentos para o modelo representado.

Podemos supor que, para uma melhor flexibilidade, a escolha dos tipos de atributos que deverão ser calculados poderia ser uma funcionalidade da classe `CIntObj` em vez de uma funcionalidade da classe `CIntManager`, como foi implementado. Por outro lado, verificamos que os modelos que requerem a geração de diferentes conjuntos de tipos de atributos produzem, internamente, diferentes conjuntos de *shaders* para os estágios de processamento da arquitetura na GPU. De modo a reduzir o número de troca de estados desses *shaders* e com isso aumentar o desempenho, cada instância de `CIntManager` processa todos os modelos (*i.e.*, instâncias de `CIntObj`) que utilizam os mesmos tipos de atributos. Se essa estratégia não fosse adotada, os modelos em `CIntManager` não seriam ordenados por atributos e trocas redundantes de estados poderiam ocorrer.

A utilização de *shaders* na biblioteca requer a existência de *hardware* gráfico compatível com o modelo de *shader* 3.0. Isto é necessário porque os estágios de modificação de atributos de vértices utilizam amostragem de texturas no *shader* de vértices, característica esta suportada apenas a partir deste modelo. Além disso, este é o primeiro modelo de *shader* a suportar a quantidade de instruções e registradores utilizada nos *shaders* de estimativa de elementos de geometria diferencial.

No restante deste capítulo, apresentamos a sintaxe e descrição de cada um dos comandos implementados na biblioteca.

C.1 Conjunto de funções de interação

De modo a integrar a arquitetura de interação às atuais APIs gráficas, propomos um conjunto de aproximadamente 50 comandos divididos entre as classes `CIntObj` e `CIntManager`. Descrevemos a seguir os comandos implementados na biblioteca para para realizar as seguintes tarefas da arquitetura proposta:

- Enviar os dados necessários à execução da arquitetura de interação, segundo a interface de entrada descrita na seção 5.4.2;
- Executar os estágios de processamento da arquitetura descritos na seção 5.4;
- Receber os resultados do processamento segundo a interface de saída descrita na seção 5.4.3.

Essas funções são detalhadas a seguir.

C.1.1 Funções de interface de entrada

A interface de entrada encerra a maior parte dos comandos da biblioteca, e podem ser divididas em funções de configuração da região de interesse, funções de gerenciamento dos objetos sob interação (criação e destruição de objetos), funções de configuração dos atributos a serem processados e funções de configuração dos modelos geométricos correspondentes aos objetos sob interação (definição da geometria dos modelos, definição das funções de chamada de retorno e definição dos *shaders* de deformação).

Funções de configuração da região de interesse

- Sintaxe: `CIntManager::SetROI( int iLeft, int iTop, int iRight, int iBottom ).`

  Descrição: informa as coordenadas de um retângulo que representa a região de interesse na qual os atributos devem ser calculados para cada *pixel*. A região de interesse é descrita em coordenadas relativas à janela de visualização.

  Internamente, a arquitetura utiliza essas coordenadas para modificar as configurações da janela de visão e matriz de projeção quando os modelos geométricos estão sendo renderizados nos *buffers* de renderização não visíveis. A alteração das configurações da janela de visão e matriz de projeção demonstrou ser melhor do que usar o *scissor test*, uma vez que os *buffers* de renderização podem ser gerados com o tamanho exato da região de interesse, e portanto resultando em economia de memória de vídeo.

  Parâmetros:

  - `iLeft`, `iTop`: coordenadas do canto superior esquerdo da região de interesse, em coordenadas relativas ao sistema de referência da janela.
  - `iRight`, `iBottom`: coordenadas do canto inferior direito da região de interesse, em coordenadas relativas ao sistema de referência da janela.

- Sintaxe: `CIntManager::GetROI( int &iLeft, int &iTop, int &iRight, int &iBottom ).`

  Descrição: retorna, nos parâmetros passados como referência, as coordenadas atuais da região de interesse. Os parâmetros são os mesmos de `CIntManager::SetROI()`.

- Sintaxe: `CIntManager::SetZNear( float fZNear ).`

Descrição: informa a distância do plano de recorte próximo (*near clipping plane*). Este valor é utilizado pela arquitetura para calcular a matriz de projeção utilizada internamente na definição da região de interesse. Deve ser o mesmo valor utilizado na visualização da cena.

Parâmetros:

- `fZNear`: distância ao longo do eixo Z do plano de recorte próximo, em coordenadas relativas ao sistema de referência da câmera.

- Sintaxe: `CIntManager::GetZNear()`.

  Descrição: retorna o valor atual da distância do plano de recorte próximo.

  Valor de retorno: distância atual informada com `CIntManager::SetZNear()`.

- Sintaxe: `CIntManager::SetZFar( float fZfar )`.

  Descrição: informa a distância do plano de recorte distante (*far clipping plane*). Esse valor é utilizado pela arquitetura para calcular a matriz de projeção utilizada internamente na definição da região de interesse. Deve ser o mesmo valor utilizado na visualização da cena.

  Parâmetros:

  - `fZfar`: distância ao longo do eixo Z do plano de recorte distante, em coordenadas relativas ao sistema de referência da câmera.

- Sintaxe: `CIntManager::GetZFar()`.

  Descrição: retorna o valor atual da distância do plano de recorte distante.

  Valor de retorno: distância atual informada com `CIntManager::SetZFar()`.

Funções de gerenciamento de objetos sob interação

- Sintaxe: `CIntManager::CreateObject( CIntObj **pObj,`
  `const char *szAdjWriteFile = NULL,`
  `const char *szAdjReadFile = NULL )`.

  Descrição: cria uma instância de `CIntObj` que será utilizada pela arquitetura segundo as configurações de atributos da instância `CIntManager` que a chama. Essa função deve ser chamada para cada modelo sob interação. A configuração de cada `CIntObj` se dá através de suas funções-membro.

  Parâmetros:

  - `pObj`: ponteiro para um ponteiro de `CIntObj` que receberá o novo objeto criado.
  - `szAdjWriteFile`: nome do arquivo no qual os dados pré-computados de vizinhança de 1-anel serão armazenados. Este parâmetro é opcional e é `NULL` por *default*.
  - `szAdjReadFile`: nome do arquivo do qual os dados pré-computados de vizinhança de 1-anel serão lidos. Este parâmetro é opcional e é `NULL` por *default*. Entretanto, o uso desse parâmetro aumenta o desempenho durante a criação dos objetos uma vez que não será gasto tempo para realizar o pré-processamento das estruturas de dados (figura 4.14).

- Sintaxe: `CIntManager::DestroyObject( CIntObj *pObj ).`

  Descrição: destrói uma instância de `CIntObj` criada previamente através de `CIntManager ::CreateObject().`

  Parâmetros:

  - `pObj`: ponteiro para o objeto `CIntObj` que será destruído.

- Sintaxe: `CIntManager::GetNumObjects().`

  Descrição: retorna o número atual de objetos gerenciados pela instância de `CIntManager`, *i.e.*, o número de objetos criados através de `CIntManager::CreateObject().`

  Valor de retorno: número atual de objetos criados pela instância de `CIntManager`.

- Sintaxe: `CIntManager::GetObjects( CIntObj **pObjList ).`

  Descrição: obtém a lista de ponteiros de instâncias de `CIntObj` atualmente gerenciadas pela instância de `CIntManager`.

  Parâmetros:

  - `pObjList`: ponteiro para um arranjo que será preenchido com os ponteiros de instâncias de `CIntObj` atualmente gerenciadas. O número de ponteiros retornados pode ser obtido de `CIntManager::GetNumObjects().`

Funções de configuração de atributos

- Sintaxe: `CIntManager::SetAttributes( ATTTYPE *pAttributes ).`

  Descrição: informa quais os tipos de atributos que deverão ser processados pela arquitetura segunda a instância atual de `CIntManager`. Isso inclui tanto os atributos já existentes no *buffer* de vértices do modelo original, como atributos que devem ser calculados na GPU. O número de atributos selecionados é inversamente proporcional ao desempenho da arquitetura.

Na etapa de codificação, a arquitetura procura realizar o processamento de todos os atributos em um único passo de renderização. Para isso é utilizada a funcionalidade de escrita simultânea em múltiplos alvos de renderização (*multiple render targets*). Entretanto, se o número desses alvos em um único passo não for suficiente para armazenar todos os atributos processados, a arquitetura automaticamente executará passos de renderização adicionais.

Parâmetros:

- `pAttributes`: ponteiro para um arranjo de tipos de atributos que devem ser processados pela arquitetura. Cada tipo de atributo é definido por um valor de enumeração `ATTTYPE`, que pode ser: `ATTTYPE_DEPTH` para o valor de profundidade em coordenadas normalizadas do dispositivo; `ATTTYPE_TEXCOORD` para coordenadas de textura; `ATTTYPE_TBN` para uma base tangente composta de vetor normal, vetor tangente e vetor bitangente; `ATTTYPE_USERDEFI` para um valor definido pela aplicação para geometria indexada; `ATTYPE_USERDEFNI` para um valor definido pela aplicação para geometria não indexada; `ATTYPE_NORMAL` para o vetor normal; `ATTTYPE_CURV` para as curvaturas principais e direções principais; `ATTTYPE_TENSOR2` para os coeficientes do tensor de curvatura; `ATTTYPE_TENSOR3` para os coeficientes do tensor de derivada de curvatura. O arranjo é finalizado com o valor `ATTTYPE_END`. Se nenhum dos atributos `ATTTYPE_TBN`, `ATTYPE_NORMAL`, `ATTTYPE_CURV`, `ATTTYPE_TENSOR2` e `ATTTYPE_TENSOR3` forem especificados, o estágio 2 de processamento da arquitetura será desabilitado automaticamente (ver figuras 5.9 e 5.10).

- Sintaxe: `CIntManager::BindSemantics( SEMANTICBINDING pBindings ).`

  Descrição: informa a ligação semântica entre cada atributo do *buffer* de vértices do modelo original (segundo a lista de semânticas definidas pela API gráfica) e cada atributo a ser processado pela arquitetura de interação.

  Parâmetros: `pBindings`: arranjo de estruturas do tipo `SEMANTICBINDING`. Cada estrutura contém duas variáveis-membro: `iVSSemantic` e `iIntSemantic`, que correspondem a valores das enumerações `VSSEMANTIC` e `INTSEMANTIC`, respectivamente. Cada ligação semântica é um par de valores dessas enumerações, e identifica qual atributo do *buffer* de vértices do modelo original corresponde a qual atributo a ser processado pela arquitetura de interação. O valor da enumeração `VSSEMANTIC` define a semântica do atributo de entrada do ponto de vista da API gráfica, podendo ser `VSSEMANTIC_NORMAL` para o vetor normal, `VSSEMANTIC_COLOR0` e `VSSEMANTIC_COLOR1` para os valores de cor primária e secundária, e `VSSEMANTIC_TEXCOORD0` até `VSSEMANTIC_TEXCOORD7` para um conjunto de até oito valores de coordenadas de textura. O valor da enumeração `INTSEMANTIC`

define a semântica do atributo do ponto de vista da arquitetura de interação, podendo ser `INTSEMANTIC_TEXCOORD` para indicar as coordenadas de textura utilizadas no cálculo das bases tangentes, `INTSEMANTIC_VERTEXID` para indicar o atributo que contém o identificador de cada vértice (atributo necessário para estimar os elementos de geometria diferencial e atualizar a geometria do modelo), `INTSEMANTIC_USERDEFI` para indicar um valor definido pela aplicação para geometria indexada, e `INTSEMANTIC_USERDEFNI` para indicar um valor definido pela aplicação para geometria não indexada. O arranjo é finalizado com uma estrutura `SEMANTICBINDING` especial chamada de `SEMANTIC_END`.

Funções de configuração dos modelos

- Sintaxe: `CIntObj::SetVertexBuffer( float *pVtPos, float pVtTexCoord, float *pVtNormal, int iSize )`.

  Descrição: informa os atributos dos vértices da geometria original, em particular a posição 3D, as coordenadas de textura e o vetor normal.

  Parâmetros:

  - `pVtPos`: ponteiro a um arranjo de coordenadas 3D contendo a posição de cada vértice, em coordenadas relativas ao sistema de coordenadas local. Cada seqüência de três valores corresponde às coordenadas de um vértice.
  - `pVtTexCoord`: ponteiro a um arranjo de coordenadas de textura. Cada seqüência de dois valores corresponde a um par de coordenadas UV de um vértice. Este parâmetro é opcional (*i.e.*, pode ser `NULL`), exceto quando a estimativa de propriedades de geometria diferencial envolva o cálculo de bases tangentes.
  - `pVtNormal`: ponteiro a um arranjo de vetores normais. Cada seqüência de três valores corresponde a às coordenadas do vetor normal de um vértice. Este parâmetro é opcional.
  - `iSize`: número de vértices da geometria.

- Sintaxe: `CIntObj::SetIndexBuffer( int *pFaceIdx, int iSize )`.

  Descrição: informa o *buffer* de índices aos vértices do *buffer* de vértices. Os índices serão utilizados para calcular a vizinhança de 1-anel de cada vértice.

  Parâmetros:

  - `pFaceIdx`: ponteiro ao *buffer* de índices. Cada elemento do *buffer* de índices é um índice numérico a um vértice contido no *buffer* de vértices informado com `CIntObj::`

`SetVertexBuffer()`. Cada seqüência de três índices corresponde a um triângulo da geometria.

– `iSize`: número de triângulos da geometria.

- Sintaxe: `CIntObj::SetPreVertexDeform( char *szFunc ).`

  Descrição: informa o *shader* de modificação de atributos de vértices que será utilizado no estágio 1 de processamento da arquitetura (ver figuras 5.9 e 5.10), *i.e.*, antes da estimativa de propriedades de geometria diferencial.

  Parâmetros:

  – `szFunc`: ponteiro para uma *string* de texto contendo o código da função de modificação de atributos de vértices, segundo a linguagem de *shaders* utilizada pela API gráfica (GLSL para OpenGL e HLSL para Direct3D). Tal função deve ter obrigatoriamente o nome `fPreVSDeform`. Em HLSL, ela deve receber como parâmetro de entrada uma estrutura de dados do tipo `VSDEFORM`. Internamente, a arquitetura define essa estrutura segundo os atributos de vértices da geometria original mapeados pela função `CIntManager::BindSemantics()`. Essa estrutura contém os atributos da geometria original de cada vértice.

    O conteúdo de `VSDEFORM` inclui necessariamente uma variável `vec4 vPos` (GLSL) ou `float4 vPos` (HLSL) contendo o atributo de posição do vértice. As demais variáveis são acrescentadas de acordo com os valores de `VSSEMANTIC` utilizados na última chamada de `CIntManager::BindSemantics()`. Por exemplo, o valor de enumeração `VSSEMANTIC_NORMAL` adiciona à estrutura a variável `float4 vNormal : NORMAL` em HLSL e `vec4 vNormal` em GLSL. O valor `VSSEMANTIC_COLOR0` adiciona a variável `float4 vColor0 : COLOR0` em HLSL e `vec4 vColor0` em GLSL. O valor `VSSEMANTIC_TEXCOORD0` adiciona a variável `float4 vTex0 : TEXCOORD0` em HLSL e `vec4 vTex0` em GLSL, e assim por diante.

    Em GLSL, `fPreVSDeform` não contém `VSDEFORM` como parâmetro, e os atributos dos vértices podem ser acessados usando a interface padrão do GLSL. Por exemplo, a posição do vértice é obtida de `gl_Vertex`.

    A função `fPreVSDeform` pode realizar modificações arbitrárias dos atributos de `VSDEFORM`. O resultado deve ser armazenado em uma nova estrutura `VSDEFORM` que deverá ser retornada pela função, seja ela em HLSL ou GLSL. Internamente, a arquitetura utilizará esses valores para atualizar as texturas que contém os dados da geometria. Esses dados serão utilizadas na estimativa das propriedades de geometria diferencial.

Se esse parâmetro é `NULL`, o estágio de modificação de atributos de vértices é desabilitado.

- Sintaxe: `CIntObj::SetPostVertexDeform( char *szFunc ).`

  Descrição: informa o *shader* de modificação de atributos de vértices que será utilizado no estágio 3 de processamento da arquitetura (ver figuras 5.9 e 5.10), *i.e.*, depois da estimativa de propriedades de geometria diferencial.

  Parâmetros:

  - `szFunc`: ponteiro para uma *string* de texto contendo o código da função de modificação de atributos de vértices, segundo a linguagem de *shaders* utilizada pela API gráfica (GLSL para OpenGL e HLSL para Direct3D). Tal função deve ter obrigatoriamente o nome `fPostVSDeform` e deve receber como parâmetro de entrada uma estrutura de dados do tipo `PSDEFORM`. Essa estrutura contém as variáveis presentes em `VSDEFORM`, além de variáveis correspondentes às propriedades de geometria diferencial estimadas no estágio 2. Por exemplo, se o vetor normal é estimado, `PSDEFORM` conterá a variável `float3 vTN : TEXCOORD9` em HLSL e `vec3 vTN` em GLSL. A estimativa de vetores tangente e bitangente adicionará as variáveis `float3 vTT : TEXCOORD10` e `float3 vTB : TEXCOORD11` em HLSL (`vec3 vTT` e `vec3 vTB` em GLSL). A estimativa de curvaturas principais e direções principais adicionará as variáveis `float4 vP1K1 : TEXCOORD12` e `float4 vP2K2 : TEXCOORD13` em HLSL (`vec4 vP1K1`, `vec4 vP2K2` em GLSL), que contém as direções principais nos três primeiros valores, e a curvatura principal no quarto valor. A estimativa dos coeficientes do tensor de curvatura adicionará a variável `float3 vTensor2 : TEXCOORD14` em HLSL (`vec3 vTensor3` em GLSL), contendo os coeficientes (e, f, g) do tensor. Por fim, a estimativa dos coeficientes do tensor de derivada de curvatura adicionará a variável `float4 vTensor3 : TEXCOORD15` em HLSL (`vec4 vTensor3` em GLSL), contendo os coeficientes (a, b, c, d) do tensor.

    A função `fPostVSDeform` pode realizar modificações arbitrárias dos atributos de `PSDEFORM`. O resultado deve ser armazenado em uma nova estrutura `PSDEFORM` que deverá ser retornada pela função. Internamente, esses atributos serão interpolados para os fragmentos contidos em cada primitiva.

    Se esse parâmetro é `NULL`, o estágio de modificação de atributos de vértices após a estimativa de propriedades de geometria diferencial é desabilitado.

- Sintaxe: `CIntObj::SetPixelDeform( char *szFunc ).`

Descrição: informa o *shader* de modificação de atributos de fragmentos que será utilizado no estágio 4 de processamento da arquitetura (ver figuras 5.9 e 5.10).

  - `szFunc`: *string* de texto contendo o código da função de modificação de atributos de fragmentos segundo a linguagem de *shaders* utilizada pela API gráfica (GLSL para OpenGL e HLSL para Direct3D). Tal função deve ter obrigatoriamente o nome `fPSDeform` e deve receber como parâmetro de entrada uma estrutura de dados do tipo `PSDEFORM`. Essa estrutura é a mesma utilizada em `CIntObj::SetPostVertexDeform()`. A função `fPSDeform` também deve retornar uma estrutura de dados do tipo `PSDEFORM` contendo os atributos modificados.

- Sintaxe: `CIntObj::TogglePreVertexDeform( bool bEnable )`.

  Descrição: habilita ou desabilita o uso da função de modificação de atributos de vértices da geometria original.

  Parâmetros:

  - `bEnable`: valor booleano que indica se a função de modificação deve ser habilitada ou não.

- Sintaxe: `CIntObj::TogglePostVertexDeform( bool bEnable )`.

  Descrição: habilita ou desabilita o uso da função de modificação de atributos de vértices da geometria após a estimativa de propriedades de geometria diferencial.

  Parâmetros:

  - `bEnable`: valor booleano que indica se a função de modificação deve ser habilitada ou não.

- Sintaxe: `CIntObj::TogglePixelDeform( bool bEnable )`.

  Descrição: habilita ou desabilita o uso da função de modificação de atributos de fragmentos.

  Parâmetros:

  - `bEnable`: valor booleano que indica se a função de modificação deve ser habilitada ou não.

- Sintaxe: `CIntObj::UsePreVertexDeform()`.

  Descrição: retorna um valor indicando se a função de modificação de atributos de vértices da geometria original está habilitada.

  Valor de retorno: valor booleano que indica se a função de modificação está habilitada ou não.

- Sintaxe: `CIntObj::UsePostVertexDeform()`.

  Descrição: retorna um valor indicando se a função de modificação de atributos de vértices da geometria após a estimativa de propriedades de geometria diferencial está habilitada.

  Valor de retorno: valor booleano que indica se a função de modificação está habilitada ou não.

- Sintaxe: `CIntObj::UsePixelDeform()`.

  Descrição: retorna um valor indicando se a função de modificação de atributos de fragmentos está habilitada.

  Valor de retorno: valor booleano que indica se a função de modificação está habilitada ou não.

- Sintaxe: `CIntObj::ToggleVisible( bool bVisible )`.

  Descrição: habilita ou desabilita a renderização do modelo na próxima chamada a `CIntManager::Render()`.

  Parâmetros:

  - `bVisible`: valor booleano que indica se a renderização do modelo deve ser habilitada ou não.

- Sintaxe: `CIntObj::IsVisible()`.

  Descrição: retorna um valor valor indicando se a renderização do modelo está habilitada.

  Valor de retorno: valor booleano que indica se a renderização do modelo está habilitada ou não.

- Sintaxe: `CIntObj::SetUpdateCallback( void fUpdate( CIntObj *pObj, void *pUserData ) )`.

  Descrição: informa a função de chamada de retorno utilizada durante o passo de renderização responsável pela atualização dos atributos de vértices da geometria, *i.e*, o estágio 1 da arquitetura.

  Parâmetros:

  - `fUpdate`: ponteiro para a função de chamada de retorno. A função deve receber como parâmetros a instância de `CIntObj` do modelo que está sendo atualizado, e um ponteiro a um tipo de dado definido pela aplicação. Os dados definidos pela aplicação são fornecidos através da função `CIntObj::SetUserData()`.

- Sintaxe: `CIntObj::SetRenderCallback( void fIdx( CIntObj *pObj, void *pUserData ), void fNonIdx( CIntObj *pObj, void *pUserData ) )`.

Descrição: informa as funções de chamada de retorno utilizadas na renderização dos atributos da geometria nos *buffers* de renderização não visíveis.

Parâmetros:

- `fIdx`: ponteiro para a função de chamada de retorno utilizada na renderização de geometria indexada. A função deve receber como parâmetros a instância de `CIntObj` do modelo que está sendo renderizado, e um ponteiro a um tipo de dado definido pela aplicação.
- `vNonIdx`: ponteiro para a função de chamada de retorno que será utilizada na renderização de geometria não indexada. Se geometria não indexada não for utilizada, este parâmetro pode ser `NULL`.

- Sintaxe: `CIntObj::GetSemantic( INTSEMANTIC iIntSemantic, VSSEMANTIC &iVsSemantic, int iIdx = 0 )`.

  Descrição: obtém o valor de semântica `VSSEMANTIC` mapeado a uma semântica `INTSEMANTIC`.

  Parâmetros:

  - `iIntSemantic`: valor de semântica `INTSEMANTIC` cujo correspondente `VSSEMANTIC` está sendo procurado.
  - `iVsSemantic`: referência ao valor de semântica `VSSEMANTIC` mapeado a `iIntSemantic`.
  - `iIdx` Quando há múltiplos valores com a mesma semântica `INTSEMANTIC`, este valor corresponde ao índice (a partir de zero) da semântica cujo mapeamento está sendo consultado.

- Sintaxe: `CIntObj::GetSemanticStr( VSSEMANTIC iVtSemantic, char *szResult )`.

  Descrição: obtém a *string* de texto correspondente a um dado valor de enumeração `VSSEMANTIC`.

  Parâmetros:

  - `iVtSemantic`: valor de enumeração `VSSEMANTIC` cuja *string* está sendo solicitada.
  - `szResult`: ponteiro a *string* de texto resultante. Tal *string* é simplesmente o nome do valor de enumeração `VSSEMANTIC`, porém sem o trecho de texto "`VSSEMANTIC_`." Por

exemplo, se `iVtSemantic` contém o valor de enumeração `VSSEMANTIC_NORMAL`, `szResult` conterá a string "`NORMAL`."

- Sintaxe: `CIntObj::GetParamStr( VSSEMANTIC iVtSemantic, char *szResult )`.

  Descrição: obtém a *string* de texto contendo o nome da variável-membro de `VSDEFORM` correspondente a um dado valor de enumeração `VSSEMANTIC`.

  Parâmetros:

  - `iVtSemantic`: valor de enumeração `VSSEMANTIC` que está sendo solicitada.
  - `szResult`: ponteiro a *string* de texto resultante. Para o valor de enumeração `VSSEMANTIC_NORMAL`, essa *string* será "`vNormal`". Para os valores `VSSEMANTIC_COLOR0` e `VSSEMANTIC_COLOR1` a *string* será "`vColor0`" e "`vColor1`", respectivamente. Para o valor `VSSEMANTIC_TEXCOORD0`, a string será "`vTex0`", e assim por diante.

- Sintaxe: `CIntObj::SetUserData( void *pUserData )`.

  Descrição: informa os dados definidos pela aplicação e que serão enviados às funções de chamada de retorno definidas com `CIntObj::SetRenderCallback()` e `CIntObj::SetUpdateCallback()`.

  Parâmetros:

  - `pUserData`: ponteiro para os dados definidos pelo aplicação.

- Sintaxe: `CIntObj::GetUserData()`.

  Descrição: retorna o ponteiro para os dados definidos pela aplicação através da função `CIntObj::SetUserData()`.

  Valor de retorno: ponteiro para os dados definidos pela aplicação.

C.1.2 Funções de processamento no laço de renderização

- Sintaxe: `CIntManager::Render()`.

  Descrição: executa o fluxo completo de processamento da arquitetura de interação de modo a modificar os atributos de cada modelo, estimar as propriedades de geometria diferencial e codificar os atributos em *buffers* de renderização não visíveis. Esse processamento é realizado para cada modelo vinculado a um objeto `CIntObj` criado através

da função `CIntManager::CreateObject()`. Neste processo de renderização, o *buffer* de profundidade utilizado é o mesmo utilizado para a visualização. O teste de profundidade é habilitado e o conteúdo deste *buffer* é alterado para cada modelo renderizado. A tarefa de limpar o conteúdo do *buffer* de profundidade é de responsabilidade da aplicação.

C.1.3 Funções de interface de saída

- Sintaxe: `CIntManager::Decode( float *pAttList[ATTTYPE_SIZE] )`.

  Descrição: obtém os atributos armazenados nos *buffers* de renderização não visíveis após a execução de `CIntManager::Render()`.

  Parâmetros:

  - `pAttList`: conjunto de `ATTTYPE_SIZE` arranjos, onde `ATTTYPE_SIZE` é o número máximo de atributos que podem ser obtidos pela arquitetura (atualmente esse valor é igual a 9). Somente os arranjos com índices `ATTTYPE` informados através de `CIntManager::SetAttributes()` serão válidos. Cada arranjo é um conjunto de valores numéricos em ponto flutuante, contendo os valores dos atributo de saída. O tamanho do arranjo é definido pelo número de *pixels* da atual região de interesse, vezes o número de valores em ponto flutuante utilizado por cada *pixel*. Por exemplo, o atributo de valor de profundidade requer, para cada *pixel*, apenas um valor em ponto flutuante. Assim, para uma região de interesse de tamanho 5×5, o arranjo `pAttList[ATTTYPE_DEPTH]` terá $5 \times 5 \times 1 = 25$ valores. Em contrapartida, para o atributo de vetor normal, são necessários 3 três valores em ponto flutuante (um para cada componente da coordenada) para cada *pixel*, de modo que `pAttList[ATTTYPE_NORMAL]` terá $5 \times 5 \times 3 = 75$ valores.

Comumente, funções tais como `CIntManager::BindSemantics()`, `CIntManager::SetAttributes()`, além de praticamente todas as funções de `CIntObj`, são chamadas apenas uma vez durante a inicialização da arquitetura e inicialização dos modelos geométricos utilizados. O uso de tais funções durante o laço de renderização pode prejudicar o desempenho da aplicação, pois elas envolvem tarefas potencialmente custosas, como criação e carregamento de texturas, criação de *shaders* e determinação da vizinhança de 1-anel dos modelos. Por outro lado, funções como `CIntManager::SetROI()`, `CIntObj::TogglePreVertexDeform()`, `CIntObj::TogglePostVertexDeform()` e funções de obtenção de estados atuais podem ser executadas para cada quadro de exibição, sem prejuízo no desempenho. Por exemplo, a região de interesse pode ser modificada de acordo com os eventos de movimentação do *mouse*, devendo ser chamada durante o laço de renderização.

A classe `CIntObj` possui funções adicionais não citadas aqui. São elas as funções de obtenção de dados internos criados pela arquitetura, tais como os *shaders* utilizados em cada etapa de processamento, e as texturas de *cache* responsáveis por armazenar os atributos modificados dos vértices, e dados sobre a vizinhança de 1-anel de cada vértice do modelo.